#### MPI: A Message-Passing Interface Standard Version 4.0

Message Passing Interface Forum

June 9, 2021

1	This document describes the Message-Passing Interface (MPI) standard, version 4.0.
2	The MPI standard includes point-to-point message-passing, collective communications, group
3	and communicator concepts, process topologies, environmental management, process cre-
4	ation and management, one-sided communications, extended collective operations, external
5	interfaces, I/O, some miscellaneous topics, and multiple tool interfaces. Language bindings
6	for C and Fortran are defined.
7	Historically, the evolution of the standard is from MPI-1.0 (May 5, 1994) to MPI-1.1
8	(June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and
9	published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functional-
10	ity, to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2
11	
12	and some errata documents to one combined document, and to MPI-2.1 (June 23, 2008),
	combining the previous documents. Version MPI-2.2 (September 4, 2009) added additional
13	clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) was an exten-
14	sion of MPI-2.2. Version MPI-3.1 (June 4, 2015) added clarifications and minor extensions
15	to MPI-3.0. Version MPI-4.0 (June 9, 2021) adds significant new features to MPI-3.1.
16	
17	Comments. Please send comments on MPI to the MPI Forum as follows:
18	
19	1. Subscribe to https://lists.mpi-forum.org/mailman/listinfo/mpi-comments
20	2. Send your comment to: mpi-comments@lists.mpi-forum.org, together with the version
21	of the MPI standard and the page and line numbers on which you are commenting.
22	Only use the official versions.
23	
24	Your comment will be forwarded to MPI Forum committee members for consideration.
25	Messages sent from an unsubscribed e-mail address will not be considered.
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	@1002 1004 1005 1006 1007 2000 2000 2012 2015 2021 University of Theorem
46	©1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, 2021 University of Tennessee, Knowille, Tennessee, Dermission to convint out for all or port of this material is granted
47	Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted,
48	provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.

ii

Version 4.0: June 9, 2021. This version of the MPI-4 Standard is a major update and includes significant new functionality. The largest changes are the addition of large-count versions of many routines to address the limitations of using an int or INTEGER for the count parameter, persistent collectives, partitioned communications, an alternative way to initialize MPI, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

 $\mathbf{5}$ 

 $^{24}$ 

 $\frac{44}{45}$ 

Version 3.1: June 4, 2015. This document contains mostly corrections and clarifications to the MPI-3.0 document. The largest change is a correction to the Fortran bindings introduced in MPI-3.0. Additionally, new functions added include routines to manipulate MPI\_Aint values in a portable manner, nonblocking collective I/O routines, and routines to get the index value by name for MPI\_T performance and control variables.

Version 3.0: September 21, 2012. Coincident with the development of MPI-2.2, the MPI Forum began discussions of a major extension to MPI. This document contains the MPI-3 Standard. This version of the MPI-3 standard contains significant extensions to MPI functionality, including nonblocking collectives, new one-sided communication operations, and Fortran 2008 bindings. Unlike MPI-2.2, this standard is considered a major update to the MPI standard. As with previous versions, new features have been adopted only when there were compelling needs for the users. Some features, however, may have more than a minor impact on existing MPI implementations.

Version 2.2: September 4, 2009. This document contains mostly corrections and clarifications to the MPI-2.1 document. A few extensions have been added; however all correct MPI-2.1 programs are correct MPI-2.2 programs. New features were adopted only when there were compelling needs for users, open source implementations, and minor impact on existing MPI implementations.

Version 2.1: June 23, 2008. This document combines the previous documents MPI-1.3 (May 30, 2008) and MPI-2.0 (July 18, 1997). Certain parts of MPI-2.0, such as some sections of Chapter 4, Miscellany, and Chapter 7, Extended Collective Operations, have been merged into the chapters of MPI-1.3. Additional errata and clarifications collected by the MPI Forum are also included in this document.

Version 1.3: May 30, 2008. This document combines the previous documents MPI-1.1 (June 12, 1995) and the MPI-1.2 chapter in MPI-2 (July 18, 1997). Additional errata collected by the MPI Forum referring to MPI-1.1 and MPI-1.2 are also included in this document.

Version 2.0: July 18, 1997. Beginning after the release of MPI-1.1, the MPI Forum began meeting to consider corrections and extensions. MPI-2 has been focused on process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O. A miscellany chapter discusses items that do not fit elsewhere, in particular language interoperability.

Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the standard "MPI-2: Extensions to the Message-Passing Interface", July 18, 1997. This section 47 contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only 48

iii

new function in MPI-1.2 is one for identifying to which version of the MPI Standard the
 implementation conforms. There are small differences between MPI-1 and MPI-1.1. There
 are very few differences between MPI-1.1 and MPI-1.2, but large differences between MPI-1.2
 and MPI-2.

<sup>6</sup> Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum
 <sup>7</sup> reconvened to correct errors and make clarifications in the MPI document of May 5, 1994,
 <sup>8</sup> referred to below as Version 1.0. These discussions resulted in Version 1.1. The changes
 <sup>9</sup> from Version 1.0 are minor. A version of this document with all changes marked is available.

<sup>11</sup> Version 1.0: May, 1994. The Message-Passing Interface Forum, with participation from
 <sup>12</sup> over 40 organizations, has been meeting since January 1993 to discuss and define a set of
 <sup>13</sup> library interface standards for message passing. The Message-Passing Interface Forum is
 <sup>14</sup> not sanctioned or supported by any official standards organization.

<sup>15</sup> The goal of the Message-Passing Interface, simply stated, is to develop a widely used <sup>16</sup> standard for writing message-passing programs. As such the interface should establish a <sup>17</sup> practical, portable, efficient, and flexible standard for message-passing.

<sup>18</sup> This is the final report, Version 1.0, of the Message-Passing Interface Forum. This <sup>19</sup> document contains all the technical features proposed for the interface. This copy of the <sup>20</sup> draft was processed by  $\text{LAT}_{\text{E}}X$  on May 5, 1994.

- $^{22}$

### Contents

Li	List of Figures xviii							
Li	st of 7	Tables					xx	
A	know	ledgments					xxii	
1	Intro	oduction to MPI						
	1.1	Overview and Goals					1	
	1.2	Background of MPI-1.0					2	
	1.3	Background of MPI-1.1, MPI-1.2, and MPI-2.0					2	
	1.4	Background of MPI-1.3 and MPI-2.1					3	
	1.5	Background of MPI-2.2					4	
	1.6	Background of MPI-3.0					4	
	1.7	Background of MPI-3.1					4	
	1.8	Background of MPI-4.0					5	
	1.9	Who Should Use This Standard?					5	
	1.10	What Platforms Are Targets for Implementation?					5	
	1.11	What Is Included in the Standard?					5	
	1.12	What Is Not Included in the Standard?					6	
	1.13	Organization of This Document					6	
2	MPI	Terms and Conventions					11	
	2.1	Document Notation					11	
	2.2	Naming Conventions					11	
	2.3	Procedure Specification					12	
	2.4	Semantic Terms					13	
		2.4.1 MPI Operations					13	
		2.4.2 MPI Procedures					$15^{-5}$	
		2.4.3 MPI Datatypes					17	
	2.5	Datatypes					18	
		2.5.1 Opaque Objects					18	
		2.5.2 Array Arguments					20	
		2.5.3 State					$\frac{-0}{20}$	
		2.5.4 Named Constants					$\frac{-0}{20}$	
		2.5.5 Choice					20	
		2.5.6 Absolute Addresses and Relative Address Displacements	•			•	$\frac{21}{22}$	
		2.5.7 File Offsets	•			•	$\frac{22}{22}$	
		2.5.8 Counts				•	$\frac{22}{22}$	
	2.6	Language Binding					23	

		2.6.1	Deprecated and Removed Interfaces
		2.6.2	Fortran Binding Issues
		2.6.3	C Binding Issues
		2.6.4	Functions and Macros
	2.7	Proces	ses
	2.8	Error 1	Handling
	2.9		nentation Issues
		2.9.1	Independence of Basic Runtime Routines
		2.9.2	Interaction with Signals
	2.10	Examp	$ples \dots \dots$
3	Poin	t-to-Pc	bint Communication 3
U	3.1		uction
	3.2		ng Send and Receive Operations
	0.4	3.2.1	Blocking Send
		3.2.1 3.2.2	Message Data
		3.2.2	Message Envelope
		3.2.3 3.2.4	
		3.2.4 3.2.5	8
		3.2.6	0
	0.0	3.2.7	Blocking Send-Receive
	3.3	•	The Matching and Data Conversion
		3.3.1	Type Matching Rules   4
			Type MPI_CHARACTER
		3.3.2	Data Conversion
	3.4		unication Modes
	3.5		tics of Point-to-Point Communication
	3.6		Allocation and Usage 5
		3.6.1	Model Implementation of Buffered Mode         6
	3.7		ocking Communication
		3.7.1	Communication Request Objects
		3.7.2	Communication Initiation
		3.7.3	Communication Completion
		3.7.4	Semantics of Nonblocking Communications
		3.7.5	Multiple Completions
		3.7.6	Non-Destructive Test of status
	3.8	Probe	and Cancel
		3.8.1	Probe
		3.8.2	Matching Probe
		3.8.3	Matched Receives
		3.8.4	Cancel
	3.9	Persist	ent Communication Requests
	3.10		$\mathbf{rocesses} \dots \dots$
4	Part	itioned	Point-to-Point Communication 10
4	<b>F a t</b> 4.1		$\mathbf{u}$ ction $\dots \dots \dots$
	4.1		tics of Partitioned Point-to-Point Communication
	1.4	4.2.1	Communication Initialization and Starting with Partitioning 10
		1.4.	- Communication intranzation and Starting with Latitudining 10

		4.2.2	Communication Completion under Partitioning	110
		4.2.3	Semantics of Communications in Partitioned Mode	111
	4.3	Partitic	oned Communication Examples	112
		4.3.1	Partition Communication with Threads/Tasks Using OpenMP 4.0	
			or later	112
		4.3.2	Send-only Partitioning Example with Tasks and OpenMP version	
			4.0 or later	113
		4.3.3	Send and Receive Partitioning Example with OpenMP version 4.0	
			or later	115
5	Date	atypes		119
J	5.1		1 Datatypes	119
	0.1	5.1.1	Type Constructors with Explicit Addresses	$119 \\ 121$
		5.1.1 5.1.2	Datatype Constructors	$121 \\ 121$
		5.1.2 5.1.3		$121 \\ 133$
		5.1.5 5.1.4	Subarray Datatype Constructor	
			Distributed Array Datatype Constructor	135
		5.1.5	Address and Size Functions	141
		5.1.6	Lower-Bound and Upper-Bound Markers	144
		5.1.7	Extent and Bounds of Datatypes	147
		5.1.8	True Extent of Datatypes	149
		5.1.9	Commit and Free	150
		5.1.10	Duplicating a Datatype	152
		5.1.11	Use of General Datatypes in Communication	153
		5.1.12	Correct Use of Addresses	156
		5.1.13	Decoding a Datatype	157
		5.1.14	Examples	165
	5.2		nd Unpack	174
	5.3	Canoni	cal MPI_PACK and MPI_UNPACK	182
6	Coll	ective C	Communication	187
	6.1	Introdu	action and Overview	187
	6.2	Comm	inicator Argument	190
		6.2.1	Specifics for Intra-Communicator Collective Operations	190
		6.2.2	Applying Collective Operations to Inter-Communicators	191
		6.2.3	Specifics for Inter-Communicator Collective Operations	192
	6.3		Synchronization	194
	6.4	Broadc		194
		6.4.1	Example using MPI_BCAST	195
	6.5			196
	0.0	6.5.1	Examples using MPI_GATHER, MPI_GATHERV	200
	6.6	Scatter		206
	0.0	6.6.1	Examples using MPI_SCATTER, MPI_SCATTERV	210
	6.7		-to-all	$210 \\ 213$
	0.1	6.7.1	Example using MPI_ALLGATHER	210
	6.8		All Scatter/Gather	210
	6.9		Reduction Operations	217
	0.0	6.9.1	Reduce	223 224
		6.9.2	Predefined Reduction Operations	224
		0.0.4		<u> </u>

		6.9.3	Signed Characters and Reductions	229
		6.9.4	MINLOC and MAXLOC	229
		6.9.5	User-Defined Reduction Operations	233
			Example of User-Defined Reduce	237
		6.9.6	All-Reduce	238
		6.9.7	Process-Local Reduction	240
	6.10	Reduce	-Scatter	242
		6.10.1	MPI_REDUCE_SCATTER_BLOCK	242
		6.10.2	MPI_REDUCE_SCATTER	244
	6.11	Scan .		245
		6.11.1	Inclusive Scan	246
		6.11.2	Exclusive Scan	247
		6.11.3	Example using MPI_SCAN	248
	6.12	Nonblo	cking Collective Operations	250
		6.12.1	Nonblocking Barrier Synchronization	252
		6.12.2	Nonblocking Broadcast	253
			Example using MPI_IBCAST	253
		6.12.3	Nonblocking Gather	254
		6.12.4	Nonblocking Scatter	257
		6.12.5	Nonblocking Gather-to-all	260
		6.12.6	Nonblocking All-to-All Scatter/Gather	263
		6.12.7	Nonblocking Reduce	269
		6.12.8	Nonblocking All-Reduce	270
		6.12.9	Nonblocking Reduce-Scatter with Equal Blocks	271
			Nonblocking Reduce-Scatter	273
		6.12.11	Nonblocking Inclusive Scan	274
			Nonblocking Exclusive Scan	275
	6.13		ent Collective Operations	276
		6.13.1	Persistent Barrier Synchronization	277
		6.13.2	Persistent Broadcast	278
		6.13.3	Persistent Gather	279
		6.13.4	Persistent Scatter	283
		6.13.5	Persistent Gather-to-all	286
		6.13.6	Persistent All-to-All Scatter/Gather	289
		6.13.7	Persistent Reduce	294
		6.13.8	Persistent All-Reduce	295
		6.13.9	Persistent Reduce-Scatter with Equal Blocks	296
			Persistent Reduce-Scatter	298
			Persistent Inclusive Scan	299
			Persistent Exclusive Scan	300
	6.14		ness	301
7			ntexts, Communicators, and Caching	311
	7.1		ction	311
		7.1.1	Features Needed to Support Libraries	311
		7.1.2	MPI's Support for Libraries	312
	7.2		Concepts	314
		7.2.1	Groups	314

		7.2.2	Contexts
		7.2.3	Intra-Communicators
		7.2.4	Predefined Intra-Communicators
	7.3	Group	Management
		7.3.1	Group Accessors
		7.3.2	Group Constructors
		7.3.3	Group Destructors
	7.4	Comm	unicator Management
		7.4.1	Communicator Accessors
		7.4.2	Communicator Constructors
		7.4.3	Communicator Destructors
		7.4.4	Communicator Info
	7.5	Motiva	ting Examples
		7.5.1	Current Practice $\#1$
		7.5.2	Current Practice $\#2$
		7.5.3	(Approximate) Current Practice $#3 \dots 349$
		7.5.4	Communication Safety Example
		7.5.5	Library Example $\#1$
		7.5.6	Library Example $#2$
	7.6		$Communication \dots \dots$
		7.6.1	Inter-Communicator Accessors
		7.6.2	Inter-Communicator Operations
		7.6.3	Inter-Communication Examples
		1.0.0	Example 1: Three-Group "Pipeline"   362
			Example 2: Three-Group "Ring"
	7.7	Cachin	· · · ·
		7.7.1	Functionality   365
		7.7.2	Communicators
		7.7.3	Windows
		7.7.4	Datatypes
		7.7.5	Error Class for Invalid Keyval
		7.7.6	Attributes Example
	7.8		g Objects
	7.9		izing the Loosely Synchronous Model
	1.5	7.9.1	Basic Statements
		7.9.2	Models of Execution
		1.5.2	Static Communicator Allocation
			Dynamic Communicator Allocation
			The General Case
8	Proc	ess Top	pologies 389
	8.1	_	action
	8.2	Virtual	l Topologies
	8.3		ding in MPI
	8.4	Overvi	ew of the Functions
	8.5	Topolo	gy Constructors
		8.5.1	Cartesian Constructor
		8.5.2	Cartesian Convenience Function: MPI_DIMS_CREATE

		8.5.3	Graph Constructor	394
		8.5.4	Distributed Graph Constructor	396
		8.5.5	Topology Inquiry Functions	403
		8.5.6	Cartesian Shift Coordinates	412
		8.5.7	Partitioning of Cartesian Structures	413
		8.5.8	Low-Level Topology Functions	414
	8.6	-	orhood Collective Communication	416
		8.6.1	Neighborhood Gather	417
		8.6.2	Neighbor Alltoall	422
	8.7	Nonblo	cking Neighborhood Communication	429
		8.7.1	Nonblocking Neighborhood Gather	429
		8.7.2	Nonblocking Neighborhood Alltoall	432
	8.8	Persiste	ent Neighborhood Communication	437
		8.8.1	Persistent Neighborhood Gather	438
		8.8.2	Persistent Neighborhood Alltoall	441
	8.9	An Ap	plication Example	445
9			nmental Management	451
	9.1	-	nentation Information	451
		9.1.1	Version Inquiries	451
		9.1.2	Environmental Inquiries	453
			Tag Values	453
			Host Rank	453
			IO Rank	454
			Clock Synchronization	454
			Inquire Processor Name	454
	9.2	Memor	y Allocation	455
	9.3	Error H	Handling	458
		9.3.1	Error Handlers for Communicators	461
		9.3.2	Error Handlers for Windows	463
		9.3.3	Error Handlers for Files	465
		9.3.4	Error Handlers for Sessions	466
		9.3.5	Freeing Errorhandlers and Retrieving Error Strings	468
	9.4	Error (	Codes and Classes	469
	9.5	Error (	Classes, Error Codes, and Error Handlers	473
	9.6	Timers	and Synchronization	477
10	The	Info Ob	oject	479
11			tialization, Creation, and Management	487
	11.1		iction	487
	11.2		orld Model	488
		11.2.1	Starting MPI Processes	488
		11.2.2	Finalizing MPI	494
		11.2.3	Determining Whether MPI Has Been Initialized When Using the	
			World Model	497
		11.2.4	Allowing User Functions at MPI Finalization	498
	11.3	The Se	ssions Model	499

	11.3.1	Session Creation and Destruction Methods	501
	11.3.2	Processes Sets	503
	11.3.3	Runtime Query Functions	504
	11.3.4	Sessions Model Examples	507
11.4	Commo	on Elements of Both Process Models	513
	11.4.1	MPI Functionality that is Always Available	513
	11.4.2	Aborting MPI Processes	514
11.5	Portabl	e MPI Process Startup	
11.6		d Threads	517
	11.6.1	General	
	11.6.2	Clarifications	518
11.7		namic Process Model	521
	11.7.1	Starting Processes	521
	11.7.2	The Runtime Environment	521
11.8		Manager Interface	$521 \\ 522$
11.0	11.8.1	Processes in MPI	
	11.8.2	Starting Processes and Establishing Communication	
	11.8.3	Starting Multiple Executables and Establishing Communication	
	11.8.4	Reserved Keys	$527 \\ 530$
	11.8.5	Spawn Example	
11.0			
11.9		shing Communication	533 524
	11.9.1	Names, Addresses, Ports, and All That	534
	11.9.2	Server Routines	
	11.9.3	Client Routines	
	11.9.4	Name Publishing	
	11.9.5	Reserved Key Values	541
11 10	11.9.6	Client/Server Examples	541
11.10		Functionality	544
		Universe Size	544
		Singleton MPI Initialization	544
		MPI_APPNUM	
		Releasing Connections	
	11.10.5	Another Way to Establish MPI Communication	548
12 One-	Sided C	Communications	551
12.1	Introdu	$\operatorname{ction} \ldots \ldots$	551
12.2	Initializ	ation $\ldots$	552
	12.2.1	Window Creation	553
	12.2.2	Window That Allocates Memory	556
	12.2.3	Window That Allocates Shared Memory	558
	12.2.4	Window of Dynamically Attached Memory	562
	12.2.5	Window Destruction	565
	12.2.6	Window Attributes	566
	12.2.7	Window Info	567
12.3	Commu	mication Calls	569
	12.3.1	Put	570
	12.3.2	Get	573
	12.3.3	Examples for Communication Calls	574

	12.3.4	Accumulate Functions
		Accumulate Function
		Get Accumulate Function
		Fetch and Op Function
		Compare and Swap Function
	12.3.5	Request-based RMA Communication Operations 584
12.4	Memor	y Model
12.5	Synchro	onization Calls
	12.5.1	Fence
	12.5.2	General Active Target Synchronization
	12.5.3	Lock
	12.5.4	Flush and Sync
	12.5.5	Assertions
	12.5.6	Miscellaneous Clarifications
12.6		Iandling
	12.6.1	Error Handlers
	12.6.2	Error Classes
12.7		ics and Correctness
12.1	12.7.1	Atomicity
	12.7.1 12.7.2	Ordering
	12.7.2 12.7.3	Progress
	12.7.3 12.7.4	Registers and Compiler Optimizations
12.8		les $\ldots$
12.0	ылатр	105
13 Exte	rnal In	terfaces 633
13.1		uction
13.2		lized Requests
	13.2.1	Examples
13.3	Associa	ting Information with Status
		0
14 I/O		643
14.1	Introdu	$action \dots \dots$
	14.1.1	Definitions
14.2	File Ma	anipulation
	14.2.1	Opening a File
	14.2.2	Closing a File
	14.2.3	Deleting a File
	14.2.4	Resizing a File
	14.2.5	Preallocating Space for a File
	14.2.6	Querying the Size of a File
	14.2.7	Querying File Parameters
	14.2.8	File Info
		Reserved File Hints
14.3	File Vie	ews
14.4		ccess
1 1 1 1	14.4.1	Data Access Routines
	1 1. 1. 1	Positioning
		Synchronism
		Synomonismi

		Coordination	661
		Data Access Conventions	661
	14.4.2	Data Access with Explicit Offsets	662
	14.4.3	Data Access with Individual File Pointers	669
	14.4.4	Data Access with Shared File Pointers	680
		Noncollective Operations	681
		Collective Operations	684
		Seek	687
	14.4.5	Split Collective Data Access Routines	688
14.5	File Int	eroperability	697
	14.5.1	Datatypes for File Interoperability	699
	14.5.2	External Data Representation: "external32"	701
	14.5.3	User-Defined Data Representations	704
		Extent Callback	705
		Datarep Conversion Functions	706
	14.5.4	Matching Data Representations	708
14.6	Consist	ency and Semantics	709
	14.6.1		709
	14.6.2	Random Access vs. Sequential Files	712
	14.6.3	Progress	713
	14.6.4		713
	14.6.5		713
	14.6.6	Type Matching	714
	14.6.7	Miscellaneous Clarifications	714
	14.6.8	MPI_Offset Type	714
	14.6.9	Logical vs. Physical File Layout	714
	14.6.10	File Size	715
	14.6.11	Examples	715
		Asynchronous I/O	718
14.7	I/O Err	or Handling	719
14.8	I/O Err	or Classes	720
14.9	Exampl	es	720
	14.9.1	Double Buffering with Split Collective I/O	720
	14.9.2	Subarray Filetype Constructor	723
15 Tool	Suppor	+	725
15 1001			725
$15.1 \\ 15.2$			725
10.2	15.2.1		725
	15.2.2		726
	15.2.3		726
	15.2.4	0	727
	15.2.4 15.2.5	_	728
	15.2.6		729
	10.2.0		729
		1 0	730
			730
	15.2.7		730
	10.2.1		. 50

15.3	The MF	PI Tool Information Interface
	15.3.1	Verbosity Levels
	15.3.2	Binding MPI Tool Information Interface Variables to MPI Objects
	15.3.3	Convention for Returning Strings
	15.3.4	Initialization and Finalization
	15.3.5	Datatype System
	15.3.6	Control Variables
		Control Variable Query Functions
		Handle Allocation and Deallocation
		Control Variable Access Functions
	15.3.7	Performance Variables
	10.0.1	Performance Variable Classes
		Performance Variable Query Functions
		Performance Experiment Sessions
		Handle Allocation and Deallocation
		Starting and Stopping of Performance Variables
		Performance Variable Access Functions
	15.3.8	Events
		Event Sources
		Callback Safety Requirements
		Event Type Query Functions
		Handle Allocation and Deallocation
		Handling Dropped Events
		Reading Event Data
		Reading Event Meta Data
	15.3.9	Variable Categorization
		Category Query Functions
		Category Member Query Functions
	15.3.10	Return Codes for the MPI Tool Information Interface
	15.3.11	Profiling Interface
		Ŭ
<b>16 Dep</b>	recated	Interfaces
16.1	Depreca	ated since MPI-2.0
16.2	Depreca	ated since MPI-2.2
16.3		ated since MPI-4.0
<b>17 Rem</b>	oved In	terfaces
17.1	Remove	ed MPI-1 Bindings
	17.1.1	Overview
	17.1.2	Removed MPI-1 Functions
	17.1.3	Removed MPI-1 Datatypes
	17.1.4	Removed MPI-1 Constants
	17.1.5	Removed MPI-1 Callback Prototypes
17.2		Removed MPI-1 Callback Prototypes
	C++ B	Sindings
18 Sem	C++ B antic Ch	hanges and Warnings
	C++ B antic Ch	Sindings

18.2	Additic	onal Warnings	. 789
	18.2.1	Warnings Starting in MPI-4.0	. 789
19 Lan	guage B	indings	791
19.1	Suppor	t for Fortran	. 791
	19.1.1	Overview	. 791
	19.1.2	Fortran Support Through the mpi_f08 Module	
	19.1.3	Fortran Support Through the mpi Module	
	19.1.4	Fortran Support Through the mpif.h Include File	
	19.1.5	Interface Specifications, Procedure Names, and the Profiling Interf	
	19.1.6	MPI for Different Fortran Standard Versions	
	19.1.7	Requirements on Fortran Compilers	
	19.1.8	Additional Support for Fortran Register-Memory-Synchronization	
	19.1.9	Additional Support for Fortran Numeric Intrinsic Types	
		Parameterized Datatypes with Specified Precision and Exponen	
		Range	
		Support for Size-specific MPI Datatypes	
		Communication With Size-specific Types	
	19.1.10	Problems With Fortran Bindings for MPI	
		Problems Due to Strong Typing	
		Problems Due to Data Copying and Sequence Association with Sub	
		script Triplets	
	19.1.13	Problems Due to Data Copying and Sequence Association with Vec	
		tor Subscripts	
	19.1.14	Special Constants	
		Fortran Derived Types	
		Optimization Problems, an Overview	
		Problems with Code Movement and Register Optimization	
		Nonblocking Operations	
		Persistent Operations	
		One-sided Communication	
		MPI_BOTTOM and Combining Independent Variables in Datatype	
		Solutions	
		The Fortran ASYNCHRONOUS Attribute	
		Calling MPI_F_SYNC_REG	
		A User Defined Routine Instead of MPI_F_SYNC_REG	
		Module Variables and COMMON Blocks	
		The (Poorly Performing) Fortran VOLATILE Attribute	
		The Fortran TARGET Attribute	
	19.1.18		
	19.1.19	Permanent Data Movement	
		Comparison with C	
19.2		t for Large Count and Large Byte Displacement	
19.3		ge Interoperability	
10.0	19.3.1	Introduction	
	19.3.2	Assumptions	
	19.3.3	Initialization	
	19.3.4	Transfer of Handles	

		19.3.5	Status	 843
		19.3.6	MPI Opaque Objects	 846
			Datatypes	 846
			Callback Functions	 848
			Error Handlers	 848
			Reduce Operations	 848
		19.3.7	Attributes	 849
		19.3.8	Extra-State	 853
		19.3.9	Constants	 853
		19.3.10	Interlanguage Communication	 854
٨	Tomo	na na Di	in dia an Grana and	857
A	A.1		indings Summary Values and Handles	857
	A.1	A.1.1	Defined Constants	857
		A.1.1 A.1.2		857 871
			Types	
		A.1.3	Prototype Definitions	872
			C Bindings	872
			Fortran 2008 Bindings with the mpi_f08 Module	873
		A 1 4	Fortran Bindings with mpif.h or the mpi Module	876
		A.1.4	Deprecated Prototype Definitions	878
		A.1.5	String Values	879
			Default Communicator Names	879 870
			Reserved Data Representations	879 870
			Process Set Names	879 870
			Info Keys	879
	1.0	C	Info Values	880
	A.2		ry of the Semantics of all OpRelated Routines	881
	A.3		ngs	882
		A.3.1	Point-to-Point Communication C Bindings	882
		A.3.2	Partitioned Communication C Bindings	885
		A.3.3	Datatypes C Bindings	886
		A.3.4	Collective Communication C Bindings	889
		A.3.5	Groups, Contexts, Communicators, and Caching C Bindings	897
		A.3.6	Process Topologies C Bindings	900
		A.3.7	MPI Environmental Management C Bindings	904
		A.3.8	The Info Object C Bindings	906
		A.3.9	Process Creation and Management C Bindings	906
		A.3.10	One-Sided Communications C Bindings	907
		A.3.11	External Interfaces C Bindings	910
		A.3.12	I/O C Bindings	911
		A.3.13	Language Bindings C Bindings	915
		A.3.14	Tools / Profiling Interface C Bindings	916
		A.3.15	Tools / MPI Tool Information Interface C Bindings	916
		A.3.16	Deprecated C Bindings	919
	A.4		2008 Bindings with the mpi_f08 Module	920
		A.4.1	Point-to-Point Communication Fortran 2008 Bindings	920
		A.4.2	Partitioned Communication Fortran 2008 Bindings	930
		A.4.3	Datatypes Fortran 2008 Bindings	 931

		A.4.4 A.4.5	Collective Communication Fortran 2008 Bindings	
		A.4.0	ings	961
		A.4.6	Process Topologies Fortran 2008 Bindings	969
		A.4.0 A.4.7	MPI Environmental Management Fortran 2008 Bindings	909 978
		A.4.7 A.4.8	The Info Object Fortran 2008 Bindings	978 981
		A.4.8 A.4.9	Process Creation and Management Fortran 2008 Bindings	981
		A.4.9 A.4.10	One-Sided Communications Fortran 2008 Bindings	
		A.4.10 A.4.11	External Interfaces Fortran 2008 Bindings	
		A.4.11 A.4.12	I/O Fortran 2008 Bindings	
		A.4.12 A.4.13	Language Bindings Fortran 2008 Bindings	
		A.4.13 A.4.14	Tools / Profiling Interface Fortran 2008 Bindings	
		A.4.14 A.4.15	Deprecated Fortran 2008 Bindings	
	A.5		Bindings with mpif.h or the mpi Module	
	А.9	A.5.1	Point-to-Point Communication Fortran Bindings	
		A.5.2	Partitioned Communication Fortran Bindings	
		A.5.3	Datatypes Fortran Bindings	
		A.5.4	Collective Communication Fortran Bindings	
		A.5.5	Groups, Contexts, Communicators, and Caching Fortran Bindings	
		A.5.6	Process Topologies Fortran Bindings	
		A.5.7	MPI Environmental Management Fortran Bindings	
		A.5.8	The Info Object Fortran Bindings	
		A.5.9	Process Creation and Management Fortran Bindings	
		A.5.10		
		A.5.10 A.5.11	One-Sided Communications Fortran Bindings	
		A.5.11 A.5.12		
		A.5.12 A.5.13	I/O Fortran Bindings	
		A.5.13 A.5.14	Language Bindings Fortran Bindings	
		A.5.14 A.5.15		
			Deprecated Fortran Bindings	
В		nge-Log		1045
	B.1	<u> </u>	s from Version 3.1 to Version 4.0	
		B.1.1	Fixes to Errata in Previous Versions of MPI	
		B.1.2	Changes in MPI-4.0	
	B.2	<u> </u>	s from Version 3.0 to Version 3.1	
		B.2.1	Fixes to Errata in Previous Versions of MPI	
		B.2.2	Changes in MPI-3.1	
	B.3	Changes	s from Version 2.2 to Version 3.0	
		B.3.1	Fixes to Errata in Previous Versions of $MPI$	
		B.3.2	Changes in MPI-3.0	
	B.4	Changes	s from Version 2.1 to Version 2.2	1057
	B.5	Changes	s from Version 2.0 to Version 2.1	1060
Bi	bliog	raphy		1065
Ge	enera	Index		1071
Ex	ampl	es Inde	x	1079

MPI Constant and Predefined Handle Index	1082
MPI Declarations Index	1088
MPI Callback Function Prototype Index	1089
MPI Function Index	1091

# List of Figures

2.1	State transition diagram for blocking operations	14
2.2	State transition diagram for nonblocking operations	14
2.3	State transition diagram for persistent operations	14
6.1		189
6.2		193
6.3		193
6.4		200
6.5		202
6.6		202
6.7		203
6.8		205
6.9	1	211
6.10	*	211
	1	212
	1 1	304
6.13	Overlapping Communicators Example	308
7.1	Inter-communicator creation using MPI_COMM_CREATE	333
7.2	Inter-communicator construction with MPI_COMM_SPLIT	337
7.3	Recursive communicator creation with MPI_COMM_SPLIT_TYPE	342
7.4	Three-group pipeline	362
7.5	Three-group ring	363
8.1	Neighborhood gather communication example	419
8.2	Cartesian neighborhood allgather example for 3 and 1 processes in a dimension	
8.3	Cartesian neighborhood alltoall example for 3 and 1 processes in a dimension	424
8.4		447
8.5	Communication routine with local data copying and sparse neighborhood	
		448
8.6	Communication routine with sparse neighborhood all-to-all-w and without	
	10 0	449
8.7	Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying	450
		100
11.1	Session handle to communicator	500
11.2	Process set examples	504
12.1	Schematic description of the public/private window operations in the	
_		592

12.2	Active target communication	<i>)</i> 5
12.3	Active target communication, with weak synchronization	<del>)</del> 6
12.4	Passive target communication	<b>)</b> 7
12.5	Active target communication with several processes	)1
12.6	Symmetric communication	20
12.7	Deadlock situation	20
12.8	No deadlock	20
14.1	Etypes and filetypes	14
14.2	Partitioning a file among parallel processes	14
14.3	Displacements	
		57
14.4	Displacements	57 23

## List of Tables

2.1	Deprecated and removed constructs	25
3.1 3.2 3.3 3.4	Predefined MPI datatypes corresponding to Fortran datatypes Predefined MPI datatypes corresponding to C datatypes Predefined MPI datatypes corresponding to both C and Fortran datatypes . Predefined MPI datatypes corresponding to C++ datatypes	33 34 35 35
5.1	combiner values returned from MPI_TYPE_GET_ENVELOPE $\ldots \ldots \ldots$	159
7.1	MPI_COMM_* Function Behavior (in Inter-Communication Mode)	357
9.1 9.2	Error classes (Part 1)	$\begin{array}{c} 471\\ 472 \end{array}$
11.1	List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization $\ldots$ .	513
	C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR	$\begin{array}{c} 566 \\ 610 \end{array}$
$14.2 \\ 14.3 \\ 14.4$	Data access routines	659 702 703 703 721
$15.2 \\ 15.3 \\ 15.4 \\ 15.5 \\ 15.6$	MPI tool information interface verbosity levels	<ul> <li>732</li> <li>733</li> <li>736</li> <li>740</li> <li>760</li> <li>761</li> <li>779</li> </ul>
$17.1 \\ 17.2$	Removed MPI-1 functions and their replacements	787 788 788
	Removed MPI-1 collback prototypes and their replacements	788 788

1	19.1 Specific Fortran procedure names and related calling conventions 799
2	19.2 Occurrence of Fortran optimization problems
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	
10	

#### Acknowledgments

process not mentioned above.

This document is the product of a number of distinct efforts in four distinct phases: one for each of MPI-1, MPI-2, MPI-3, and MPI-4. This section describes these in historical order, starting with MPI-1. Some efforts, particularly parts of MPI-2, had distinct groups of individuals associated with them, and these efforts are detailed separately.

6

 $\overline{7}$ 

8

9 10 11

12

13

14

15

16

17

18

This document represents the work of many people who have served on the MPI Forum. The meetings have been attended by dozens of people from many parts of the world. It is the hard and dedicated work of this group that has led to the MPI standard.

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of the Message-Passing Interface (MPI), many people helped with this effort.

Those who served as primary coordinators in MPI-1.0 and MPI-1.1 are:

19• Jack Dongarra, David Walker, Conveners and Meeting Chairs 2021• Ewing Lusk, Bob Knighten, Minutes 22 • Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communication 23 $^{24}$ • Al Geist, Marc Snir, Steve Otto, Collective Communication 2526• Steve Otto, Editor 27• Rolf Hempel, Process Topologies 28 29 • Ewing Lusk, Language Binding 30 31• William Gropp, Environmental Management 32 • James Cownie, Profiling 33 34 • Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, 35Contexts, and Communicators 36 37 • Steven Huss-Lederman, Initial Implementation Subset 38 39 The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 40

1	Ed Anderson	Robert Babb	Joe Baron	Eric Barszcz		
2	Scott Berryman	Rob Bjornson	Nathan Doss	Anne Elster		
3	Jim Feeney	Vince Fernando	Sam Fineberg	Jon Flower		
4	Daniel Frye	Ian Glendinning	Adam Greenberg	Robert Harrison		
5	Leslie Hart	Tom Haupt	Don Heller	Tom Henderson		
6	Alex Ho	C.T. Howard Ho	Gary Howell	John Kapenga		
7	James Kohl	Susan Krauss	Bob Leary	Arthur Maccabe		
8	Peter Madams	Alan Mainwaring	Oliver McBryan	Phil McKinley		
9	Charles Mosher	Dan Nessett	Peter Pacheco	Howard Palmer		
10	Paul Pierce	Sanjay Ranka	Peter Rigsbee	Arch Robison		
11	Erich Schikuta	Ambuj Singh	Alan Sussman	Robert Tomlinson		
12	Robert G. Voigt	Dennis Weeks	Stephen Wheat	Steve Zenith		
13						
14				atory made the draft avail-		
15				distributing the document.		
16		-		RPA and NSF under grant		
17				nology Center Cooperative		
18	0	,	mmission of the Eur	opean Community through		
19	Esprit project P6643 (Pl	PPE).				
20						
21	MPI-1.2 and MPI-2.0:					
22						
23	Those who served as prin	mary coordinators in	n MPI-1.2 and MPI-	2.0 are:		
24	• Ewing Lusk, Convener and Meeting Chair					
25 26						
27	• Steve Huss-Lederman, Editor					
28 29	• Ewing Lusk, Miscellany					
30	• Bill Saphir, Process Creation and Management					
31 32	• Marc Snir, One-Sided Communications					
33	• William Gropp and Anthony Skjellum, Extended Collective Operations					
34 35	• Steve Huss-Lederm	an, External Interfa	aces			
36 37	• Bill Nitzberg, I/O					
38	• Andrew Lumsdaine	e, Bill Saphir, and J	effrey M. Squyres, I	Language Bindings		
39 40	• Anthony Skjellum	and Arkady Kanevs	ky, Real-Time			
41						
42	8		active participants v	who attended MPI-2 Forum		
43	meetings and are not me	entioned above.				
44						
45						
46						
47						
48						

	Greg Astfalk	Robert Babb	Ed Benson	Rajesh Bordawekar	1		
	Pete Bradley	Peter Brennan	Ron Brightwell	Maciej Brodowicz	2		
	Eric Brunner	Greg Burns	Margaret Cahir	Pang Chen	3		
	Ying Chen	Albert Cheng	Yong Cho	Joel Clark	4		
	Lyndon Clarke	Laurie Costello	Dennis Cottel	Jim Cownie	5		
	Zhenqian Cui	Suresh Damodaran-Kar	nal	Raja Daoud	6		
	Judith Devaney	David DiNucci	Doug Doefler	Jack Dongarra	7		
	Terry Dontje	Nathan Doss	Anne Elster	Mark Fallon	8		
	Karl Feind	Sam Fineberg	Craig Fischberg	Stephen Fleischman	9		
	Ian Foster	Hubertus Franke	Richard Frost	Al Geist	10		
	Robert George	David Greenberg	John Hagedorn	Kei Harada	11		
	Leslie Hart	Shane Hebert	Rolf Hempel	Tom Henderson	12		
	Alex Ho	Hans-Christian Hoppe	Joefon Jann	Terry Jones	13		
	Karl Kesselman	Koichi Konishi	Susan Kraus	Steve Kubica	14		
	Steve Landherr	Mario Lauria	Mark Law	Juan Leon	15		
	Lloyd Lewins	Ziyang Lu	Bob Madahar	Peter Madams	16		
	John May	Oliver McBryan	Brian McCandless	Tyce McLarty	17		
	Thom McMahon	Harish Nag	Nick Nevin	Jarek Nieplocha	18		
	Ron Oldfield	Peter Ossadnik	Steve Otto	Peter Pacheco	19		
	Yoonho Park	Perry Partow	Pratap Pattnaik	Elsie Pierce	20		
	Paul Pierce	Heidi Poxon	Jean-Pierre Prost	Boris Protopopov	21		
	James Pruyve	Rolf Rabenseifner	Joe Rieken	Peter Rigsbee	22		
	Tom Robey	Anna Rounbehler	Nobutoshi Sagawa	Arindam Saha	23		
	Eric Salo	Darren Sanders	Eric Sharakan	Andrew Sherman	24		
	Fred Shirley	Lance Shuler	A. Gordon Smith	Ian Stockdale	25		
	David Taylor	Stephen Taylor	Greg Tensa	Rajeev Thakur	26		
	Marydell Tholburn	Dick Treumann	Simon Tsang	Manuel Ujaldon	27		
	David Walker	Jerrell Watts	Klaus Wolf	Parkson Wong	28		
	Dave Wright				29		
					30		
		also acknowledges and ap	preciates the valuable	e input from people via	31		
e	-mail and in person.				32		
	<u> </u>	titutions supported the $N$	<b>IPI-2</b> effort through t	ime and travel support	33		
f	or the people listed a	bove.			34		
	Argonne National	Laboratory			35		
	Bolt, Beranek, an				36		
	California Institu				37		
	Center for Compu				38		
	Convex Computer	_			39		

German National Research Center for Information Technology

40

 $\frac{41}{42}$ 

43

44

 $45 \\ 46$ 

47

48

Hitachi Hughes Aircraft Company

General Electric Company

Digital Equipment Corporation

Dolphin Interconnect Solutions, Inc.

Edinburgh Parallel Computing Centre

Cray Research

Hewlett-Packard

1	Intel Corporation
2	International Business Machines
3	Khoral Research
4	Lawrence Livermore National Laboratory
5	Los Alamos National Laboratory
6	MPI Software Techology, Inc.
7	Mississippi State University
8	NEC Corporation
9	National Aeronautics and Space Administration
10	National Energy Research Scientific Computing Center
11	National Institute of Standards and Technology
12	National Oceanic and Atmospheric Administration
13	Oak Ridge National Laboratory
14	The Ohio State University
15	PALLAS GmbH
16	Pacific Northwest National Laboratory
17	Pratt & Whitney
18	San Diego Supercomputer Center
19	Sanders, A Lockheed-Martin Company
20	Sandia National Laboratories
21	Schlumberger
22	Scientific Computing Associates, Inc.
23	Silicon Graphics Incorporated
24	Sky Computers
25	Sun Microsystems Computer Corporation
26	Syracuse University
27	The MITRE Corporation
28	Thinking Machines Corporation
29	United States Navy
30	University of Colorado
31	University of Denver
32	University of Houston
33	University of Illinois
34	University of Maryland
35	University of Notre Dame
36	University of San Fransisco
37	University of Stuttgart Computing Center
38	University of Wisconsin
39	MPI-2 operated on a very tight budget (in reality, it had no budget when the first
40	meeting was announced). Many institutions helped the MPI-2 effort by supporting the
41	efforts and travel of the members of the MPI Forum. Direct support was given by NSF and
42	DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and
43	Esprit under project HPC Standards (21111) for European participants.
44	
45	
46	
47	

- 48
- 48

MPI-1.3 and MPI-2.1:	1		
The editors and organizers of the combined documents have been:	2 3		
• Richard Graham, Convener and Meeting Chair	4		
• Jack Dongarra, Steering Committee	5 6		
• Al Geist, Steering Committee	7		
• William Gropp, Steering Committee	8 9		
	10		
• Rainer Keller, Merge of MPI-1.3	11 12		
• Andrew Lumsdaine, Steering Committee	13		
• Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)	14 15 16		
<ul> <li>Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)</li> </ul>	10 17 18		
All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:	19 20 21		
• William Gropp, Front Matter, Introduction, and Bibliography	22		
• Richard Graham, Point-to-Point Communication	23 24		
• Adam Moody, Collective Communication	25		
• Richard Treumann, Groups, Contexts, and Communicators	26 27		
• Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communica- tions	28 29 30		
• George Bosilca, Environmental Management	31 32		
• David Solt, Process Creation and Management	33		
• Bronis R. de Supinski, External Interfaces, and Profiling	34 35		
• Rajeev Thakur, I/O	36		
• Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary	37 38		
• Rolf Rabenseifner, Deprecated Functions and Annex Change-Log	39 40		
• Alexander Supalov and Denis Nagorny, Annex Language Bindings	41		
	42		
The following list includes some of the active participants who attended MPI-2 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.	43 44		
meetings and in the c-man discussions of the citata items and are not mentioned above.	the errata items and are not mentioned above.		
	46		
	47		

#### 

1	Pavan Balaji	Purushotham V. Bangalore	Brian Barrett		
2	Richard Barrett	Christian Bell	Robert Blackmore		
3	Gil Bloch	Ron Brightwell	Jeffrey Brown		
4	Darius Buntinas	Jonathan Carter	Nathan DeBardeleben		
5	Terry Dontje	Gabor Dozsa	Edric Ellis		
6	Karl Feind	Edgar Gabriel	Patrick Geoffray		
7	David Gingold	Dave Goodell	Erez Haba		
8	Robert Harrison	Thomas Herault	Steve Hodson		
9	Torsten Hoefler	Joshua Hursey	Yann Kalemkarian		
10	Matthew Koop	Quincey Koziol	Sameer Kumar		
11	Miron Livny	Kannan Narasimhan	Mark Pagel		
12	Avneesh Pant	Steve Poole	Howard Pritchard		
13	Craig Rasmussen	Hubert Ritzdorf	Rob Ross		
14	Tony Skjellum	Brian Smith	Vinod Tipparaju		
15	Jesper Larsson Träff	Keith Underwood	11 0		
16	-				
17		nowledges and appreciates the	valuable input from people via		
18	e-mail and in person.				
19	_	s supported the MPI-2 effort t	hrough time and travel support		
20	for the people listed above.				
21	Argonne National Labora	atory			
22	Bull				
23	Cisco Systems, Inc.				
24	Cray Inc.				
25 26	The HDF Group				
20 27	Hewlett-Packard				
27	IBM T.J. Watson Research				
29	Indiana University Institut National do Recherche en Informatique et Automatique (Inria)				
30	Institut National de Recherche en Informatique et Automatique (Inria) Intel Corporation				
31	Intel Corporation Lawrence Berkeley National Laboratory				
32	Lawrence Berkeley National Laboratory				
33	Lawrence Livermore National Laboratory				
34	Los Alamos National Lab Mathworks	blatory			
35	Mellanox Technologies				
36	Microsoft				
37	Myricom				
38	NEC Laboratories Europ	e. NEC Europe Ltd.			
39	Oak Ridge National Labo	· ·			
40	The Ohio State Universit	c .			
41	Pacific Northwest Nation	0			
42	QLogic Corporation	U			
43	Sandia National Laborate	ories			
44	SiCortex				
45	Silicon Graphics Incorporated				
46	Sun Microsystems, Inc.				
47	University of Alabama at Birmingham				
48	University of Houston				

University of Illinois at Urbana-Champaign University of Stuttgart, High Performance Computing Center Stuttgart (HLRS) University of Tennessee, Knoxville University of Wisconsin	1 2 3 4
Funding for the MPI Forum meetings was partially supported by award #CCF-0816909 from the National Science Foundation. In addition, the HDF Group provided travel support for one U.S. academic.	5 6 7 8
MPI-2.2:	9 10
All chapters have been revisited to achieve a consistent MPI-2.2 text. Those who served as authors for the necessary modifications are:	11 12 13
• William Gropp, Front Matter, Introduction, and Bibliography; MPI-2.2 Chair.	14
• Richard Graham, Point-to-Point Communication and Datatypes	15 16
• Adam Moody, Collective Communication	17 18
• Torsten Hoefler, Collective Communication and Process Topologies	19
	20 21
Richard Treumann, Groups, Contexts, and Communicators	21 22
• Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications	23
• George Bosilca, Datatypes and Environmental Management	24 25
• David Solt, Process Creation and Management	26
• Bronis R. de Supinski, External Interfaces, and Profiling	27 28
• Rajeev Thakur, I/O	29
• Jeffrey M. Squyres, Language Bindings and MPI-2.2 Secretary	30 31
	32
• Rolf Rabenseifner, Deprecated Functions, Annex Change-Log, and Annex Language Bindings	33
	34 35
• Alexander Supalov, Annex Language Bindings	36
The following list includes some of the active participants who attended MPI-2 Forum	37
meetings and in the e-mail discussions of the errata items and are not mentioned above.	38
	39 40
	41
	42
	43
	44
	45
	46 47
	48

1	Davan Dala	Purushotham V. Bangalore	Brian Barrett
2	Pavan Balaji Richard Barrett	Christian Bell	Robert Blackmore
3	Gil Bloch	Ron Brightwell	Greg Bronevetsky
4	Jeff Brown	Darius Buntinas	Jonathan Carter
5	Nathan DeBardeleben	Terry Dontje	Gabor Dozsa
6	Edric Ellis	Karl Feind	Edgar Gabriel
	Patrick Geoffray		0
7 8	David Goodell	Johann George Erez Haba	David Gingold Robert Harrison
9	Thomas Herault	Marc-André Hermanns	Steve Hodson
10	Joshua Hursey	Yutaka Ishikawa	Bin Jia
10	Hideyuki Jitsumoto	Terry Jones	Yann Kalemkarian
12	Ranier Keller	Matthew Koop	Quincey Koziol
12	Manojkumar Krishnan	Sameer Kumar	Miron Livny
14	Andrew Lumsdaine	Miao Luo	Ewing Lusk
15	Timothy I. Mattox	Kannan Narasimhan	Mark Pagel
16	Avneesh Pant	Steve Poole	Howard Pritchard
17	Craig Rasmussen	Hubert Ritzdorf	Rob Ross
18	Martin Schulz	Pavel Shamis	Galen Shipman
19	Christian Siebert	Anthony Skjellum	Brian Smith
20	Naoki Sueyasu	Vinod Tipparaju	Keith Underwood
21	Rolf Vandevaart	Abhinav Vishnu	Weikuan Yu
22			
23	The MPI Forum also acknow	vledges and appreciates the va	luable input from people via
24	e-mail and in person.		
25	The following institutions su	pported the MPI-2.2 effort three	ough time and travel support
26	for the people listed above.		
27	Argonne National Laborato	rv	
28	Auburn University	- J	
29	Bull		
30	Cisco Systems, Inc.		
31	Cray Inc.		
32	Forschungszentrum Jülich		
33	Fujitsu		
34	The HDF Group		
35	Hewlett-Packard		
36	International Business Mach	nines	
37	Indiana University		
38	Institut National de Rechere	che en Informatique et Autom	atique (Inria)
39	Institute for Advanced Scien	nce & Engineering Corporation	n
40	Intel Corporation		
41	Lawrence Berkeley National	Laboratory	
42	Lawrence Livermore Nation	al Laboratory	
43	Los Alamos National Labor	atory	
44	Mathworks		
45	Mellanox Technologies		
46	Microsoft		
47	Myricom		
48	NEC Corporation		

Oak Ridge National Laboratory	1
The Ohio State University	2
Pacific Northwest National Laboratory	3
QLogic Corporation	4
RunTime Computing Solutions, LLC	5
Sandia National Laboratories	6
SiCortex, Inc.	7
Silicon Graphics Inc.	8
Sun Microsystems, Inc.	9
Tokyo Institute of Technology	10
University of Alabama at Birmingham	11
University of Houston	12 13
University of Illinois at Urbana-Champaign	13
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS) University of Tennessee, Knoxville	14
University of Tokyo	16
University of Wisconsin	17
	18
Funding for the MPI Forum meetings was partially supported by awards $\#CCF-0816909$	19
and $\#CCF-1144042$ from the National Science Foundation. In addition, the HDF Group	20
provided travel support for one U.S. academic.	21
	22
MPI-3.0:	23
MPI-3.0 is a significant effort to extend and modernize the MPI Standard.	24
The editors and organizers of the MPI-3.0 have been:	25
	26
• William Gropp, Steering Committee, Front Matter, Introduction, Groups, Contexts,	27
and Communicators, One-Sided Communications, and Bibliography	28
• Richard Graham, Steering Committee, Point-to-Point Communication, Meeting Con-	29
vener, and MPI-3.0 Chair	30
	31 32
• Torsten Hoefler, Collective Communication, One-Sided Communications, and Process	33
Topologies	34
• George Bosilca, Datatypes and Environmental Management	35
	36
• David Solt, Process Creation and Management	37
• Bronis R. de Supinski, External Interfaces and Tool Support	38
• Rajeev Thakur, I/O and One-Sided Communications	39 40
• Darius Buntinas, Info Object	41
• Darius Buntinas, Info Object	42
• Jeffrey M. Squyres, Language Bindings and MPI-3.0 Secretary	43 44
• Rolf Rabenseifner, Steering Committee, Terms and Definitions, and Fortran Bindings,	45
Deprecated Functions, Annex Change-Log, and Annex Language Bindings	46
	47
• Craig Rasmussen, Fortran Bindings	48

The following list includes some of the active participants who attended MPI-3 Forum
 meetings or participated in the e-mail discussions and who are not mentioned above.

	meetings of participated in the e in	an aboussions and who	
3			~
4	Tatsuya Abe	Tomoya Adachi	Sadaf Alam
5	Reinhold Bader	Pavan Balaji	Purushotham V. Bangalore
6	Brian Barrett	Richard Barrett	Robert Blackmore
7	Aurelien Bouteiller	Ron Brightwell	Greg Bronevetsky
8	Jed Brown	Darius Buntinas	Devendar Bureddy
9	Arno Candel	George Carr	Mohamad Chaarawi
10	Raghunath Raja Chandrasekar	James Dinan	Terry Dontje
11	Edgar Gabriel	Balazs Gerofi	Brice Goglin
12	David Goodell	Manjunath Gorentla	Erez Haba
13	Jeff Hammond	Thomas Herault	Marc-André Hermanns
14	Jennifer Herrett-Skjellum	Nathan Hjelm	Atsushi Hori
15	Joshua Hursey	Marty Itzkowitz	Yutaka Ishikawa
16	Nysal Jan	Bin Jia	Hideyuki Jitsumoto
17	Yann Kalemkarian	Krishna Kandalla	Takahiro Kawashima
18	Chulho Kim	Dries Kimpe	Christof Klausecker
19	Alice Koniges	Quincey Koziol	Dieter Kranzlmueller
20	Manojkumar Krishnan	Sameer Kumar	Eric Lantz
21	Jay Lofstead	Bill Long	Andrew Lumsdaine
22	Miao Luo	Ewing Lusk	Adam Moody
23	Nick M. Maclaren	Amith Mamidala	Guillaume Mercier
24	Scott McMillan	Douglas Miller	Kathryn Mohror
25	Tim Murray	Tomotake Nakamura	Takeshi Nanri
26	Steve Oyanagi	Mark Pagel	Swann Perarnau
27	Sreeram Potluri	Howard Pritchard	Rolf Riesen
28	Hubert Ritzdorf	Kuninobu Sasaki	Timo Schneider
29	Martin Schulz	Gilad Shainer	Christian Siebert
30	Anthony Skjellum	Brian Smith	Marc Snir
31	Raffaele Giuseppe Solca	Shinji Sumimoto	Alexander Supalov
32	Sayantan Sur	Masamichi Takagi	Fabian Tillier
33	Vinod Tipparaju	Jesper Larsson Träff	Richard Treumann
34	Keith Underwood	Rolf Vandevaart	Anh Vo
35	Abhinav Vishnu	Min Xie	Enqiang Zhou
36			

36 37

38

44

The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The MPI Forum also thanks those that provided feedback during the public comment period. In particular, the Forum would like to thank Jeremiah Wilcock for providing detailed comments on the entire draft standard.

The following institutions supported the MPI-3 effort through time and travel support
 for the people listed above.

Argonne National Laboratory

- <sup>45</sup> Bull
- <sup>46</sup> Cisco Systems, Inc.
- <sup>47</sup> Cray Inc.
- <sup>48</sup> CSCS

ETH Zurich	1
Fujitsu Ltd.	2
German Research School for Simulation Sciences	3
The HDF Group	4
Hewlett-Packard	5
International Business Machines	6
IBM India Private Ltd	7
Indiana University	8
Institut National de Recherche en Informatique et Automatique (Inria)	9
Institute for Advanced Science & Engineering Corporation	10
Intel Corporation	11
Lawrence Berkeley National Laboratory	12
Lawrence Livermore National Laboratory	13
Los Alamos National Laboratory	14
Mellanox Technologies, Inc.	15
Microsoft Corporation	16
NEC Corporation	17
National Oceanic and Atmospheric Administration, Global Systems Division	18
NVIDIA Corporation	19
Oak Ridge National Laboratory	20
The Ohio State University	21
Oracle America	22
Platform Computing	23
RIKEN AICS	24
RunTime Computing Solutions, LLC	25
Sandia National Laboratories	26
Technical University of Chemnitz	27
Tokyo Institute of Technology	28
University of Alabama at Birmingham	29
University of Chicago	30
University of Houston	31
University of Illinois at Urbana-Champaign	32
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	33
University of Tennessee, Knoxville	34
University of Tokyo	35
Even diverse for the MDI Exercise research and a set is llar source and a large second a UCCE 0.016000	36
Funding for the MPI Forum meetings was partially supported by awards $\#CCF-0816909$	37
and $\#CCF-1144042$ from the National Science Foundation. In addition, the HDF Group	38
and Sandia National Laboratories provided travel support for one U.S. academic each.	39
	40
MPI-3.1:	41
MPI-3.1 is a minor update to the MPI Standard.	42
The editors and organizers of the MPI-3.1 have been:	43
	44
• Martin Schulz, MPI-3.1 Chair	45
• William Gropp, Steering Committee, Front Matter, Introduction, One-Sided Commu-	46
• William Gropp, Steering Committee, Front Matter, Introduction, One-Sided Commu- nications, and Bibliography; Overall Editor	47
incations, and Dibilography, Overan Editor	48

1 2	,	8	nd Definitions, and Fortran Bindings, Annex Language Bindings
3 4	• Richard L. Graham, Steering Committee, Meeting Convener		
5 6	• Jeffrey M. Squyres, Language Bindings and MPI-3.1 Secretary		
7	• Daniel Holmes, Point-to-Point Communication		
8 9	• George Bosilca, Datatypes and Environmental Management		
10	• Torsten Hoefler, Collective Communication and Process Topologies		
11 12	• Pavan Balaji, Groups, Contexts, and Communicators, and External Interfaces		
13 14	<ul> <li>Jeff Hammond, The Info Object</li> </ul>		
15	• David Solt, Process Creation and Management		
16 17	• Quincey Koziol, I/O		
18 19	• Kathryn Mohror, Tool S	upport	
20	• Rajeev Thakur, One-Side	ed Communications	
21 22 23 24	The following list include meetings or participated in the	-	articipants who attended MPI Forum
25	Charles Archer	Pavan Balaji	Purushotham V. Bangalore
26	Brian Barrett	Wesley Bland	Michael Blocksome
27	George Bosilca	Aurelien Bouteiller	Devendar Bureddy
28	Yohann Burette	Mohamad Chaarawi	Alexey Cheptsov
29	James Dinan	Dmitry Durnov	Thomas Francois
30	Edgar Gabriel	Todd Gamblin	Balazs Gerofi
31	Paddy Gillies	David Goodell	Manjunath Gorentla Venkata
32	Richard L. Graham	Ryan E. Grant	William Gropp
33	Khaled Hamidouche	Jeff Hammond	Amin Hassani
34	Marc-André Hermanns	Nathan Hjelm	Torsten Hoefler
35	Daniel Holmes	Atsushi Hori	Yutaka Ishikawa
36	Hideyuki Jitsumoto	Jithin Jose	Krishna Kandalla
37	Christos Kavouklis	Takahiro Kawashima	Chulho Kim
38	Michael Knobloch	Alice Koniges	Quincey Koziol
39	Sameer Kumar	Joshua Ladd	Ignacio Laguna
40	Huiwei Lu	Guillaume Mercier	Kathryn Mohror
41	Adam Moody	Tomotake Nakamura	Takeshi Nanri
42	Steve Oyanagi	Antonio J. Pena	Sreeram Potluri
43	Howard Pritchard	Rolf Rabenseifner	Nicholas Radcliffe
44	Ken Raffenetti	Raghunath Raja	Craig Rasmussen
45	Davide Rossetti	Kento Sato	Martin Schulz
45	Sangmin Seo	Christian Siebert	Anthony Skjellum
40	Brian Smith	David Solt	Jeffrey M. Squyres
48			

	Hari Subramoni	Shinji Sumimoto	Alexander Supalov	1
	Bronis R. de Supinski	Sayantan Sur	Masamichi Takagi	2
	Keita Teranishi	Rajeev Thakur	Fabian Tillier	3
	Yuichi Tsujita	Geoffroy Vallée	Rolf vandeVaart	4
	Akshay Venkatesh	Jerome Vienne	Venkat Vishwanath	5
	Anh Vo	Huseyin S. Yildiz	Junchao Zhang	6
	Xin Zhao			7
				8
The M	PI Forum also acknowled	ges and appreciates	the valuable input from people via	9
e-mail and				10
	-	rted the MPI-3.1 effo	ort through time and travel support	11
	ple listed above.			12
	-			13
0	ne National Laboratory			14
	n University			14
	Systems, Inc.			16
$\operatorname{Cray}$				
	, The University of Edinb	ourgh		17
ETH Z	Zurich			18
Forsch	ungszentrum Jülich			19
Fujitsu	l			20
Germa	n Research School for Sin	nulation Sciences		21
The H	DF Group			22
Interna	ational Business Machines	5		23
Institu	t National de Recherche	en Informatique et $A$	Automatique (Inria)	24
Intel C	Corporation			25
Kyush	u University			26
Lawren	nce Berkeley National La	boratory		27
	nce Livermore National L	-		28
Lenovo	)	-		29
Los Al	amos National Laborator	У		30
	ox Technologies, Inc.	0		31
	oft Corporation			32
	Corporation			33
	A Corporation			34
	idge National Laboratory	r		35
	hio State University			36
	N AICS			37
	National Laboratories			38
	Advanced Computing Ce	nter		39
	Institute of Technology	11001		40
-	sity of Alabama at Birmi	ngham		41
	sity of Houston	inginain		42
	sity of Illinois at Urbana-	Champaign		43
	sity of Oregon	-Onampaign		44
		orformance Comput:	ng Contor Stuttgart (HIDC)	45
		-	ng Center Stuttgart (HLRS)	46
	sity of Tennessee, Knoxvi	ше		47
Univer	sity of Tokyo			48

1	MPI-4.0:
2 3 4	MPI-4.0 is a major update to the MPI Standard. The editors and organizers of the MPI-4.0 have been:
5	• Martin Schulz, MPI-4.0 Chair, Info Object, External Interfaces
6 7	• Richard Graham, MPI-4.0 Treasurer
8 9	• Wesley Bland, MPI-4.0 Secretary, Backward Incompatibilities
10 11	• William Gropp, MPI-4.0 Editor, Steering Committee, Front Matter, Introduction, One-Sided Communications, and Bibliography
12 13 14	• Rolf Rabenseifner, Steering Committee, Process Topologies, Deprecated Functions, Removed Interfaces, Annex Language Bindings Summary, and Annex Change-Log.
15 16	• Purushotham V. Bangalore, Language Bindings
17	• Claudia Blaas-Schenner, Terms and Conventions
18 19	• George Bosilca, Datatypes and Environmental Management
20 21	• Ryan E. Grant, Partitioned Communication
22	• Marc-André Hermanns, Tool Support
23 24	• Daniel Holmes, Point-to-Point Communication, Sessions
25 26	• Guillaume Mercier, Groups, Contexts, Communicators, Caching
27	• Howard Pritchard, Process Creation and Management
28 29	• Anthony Skjellum, Collective Communication, I/O
30 31 32 33	As part of the development of MPI-4.0, a number of working groups were established. In some cases, the work for these groups overlapped with multiple chapters. The following describes the major working groups and the leaders of those groups:
34 35	Collective Communication, Topology, Communicators Torsten Hoefler, Andrew Lumsdaine, and Anthony Skjellum
36 37	Fault Tolerance Wesley Bland, Aurélien Bouteiller, and Richard Graham
38	Hardware-Topologies Guillaume Mercier
39 40	Hybrid & Accelerator Pavan Balaji and James Dinan
41 42	Large Counts Jeff Hammond
43	Persistence Anthony Skjellum
44 45	Point to Point Communication Daniel Holmes and Richard Graham
46 47	<b>Remote Memory Access</b> William Gropp and Rajeev Thakur
48	Semantic Terms Purushotham V. Bangalore and Rolf Rabenseifner

Sessions Daniel Holmes and Howard Pritchard				
Tools Kathryn Mohror and Marc-André Hermanns       2         3       3				
The following list includes some of the a	ctive participants who attended MPI Forum	4		
meetings or participated in the e-mail discussi		5		
incomigs of participated in the c main discussion		6		
Julien Adam	Abdelhalim Amer	7		
Charles Archer	Ammar Ahmad Awan	8		
Pavan Balaji	Purushotham V. Bangalore	9		
Mohammadreza Bayatpour	Jean-Baptiste Besnard	10		
Claudia Blaas-Schenner	Wesley Bland	11		
Gil Bloch	George Bosilca	12		
Aurelien Bouteiller	Ben Bratu	13		
Alexander Calvert	Nicholas Chaimov	14		
Sourav Chakraborty	Steffen Christgau	15		
Ching-Hsiang Chu	Mikhail Chuvelev	16		
James Clark	Carsten Clauss	17		
Isaias Alberto Compres Urena	Giuseppe Congiu	18		
Brandon Cook	James Custer	19		
Anna Daly	Hoang-Vu Dang	20		
James Dinan	Matthew Dosanjh	21		
Murali Emani	Christian Engelmann	22		
Noah Evans	Ana Gainaru	23		
Esthela Gallardo	Marc Gamell Balmana	24		
Balazs Gerofi	Salvatore Di Girolamo	25		
Brice Goglin	Manjunath Gorentla Venkata	26		
Richard Graham	Ryan E. Grant	27		
Stanley Graves	William Gropp	28		
Siegmar Gross	Taylor Groves	29		
Yanfei Guo	Khaled Hamidouche	30		
Jeff Hammond	Marc-André Hermanns	31		
Nathan Hjelm	Torsten Hoefler	32		
Daniel Holmes	Atsushi Hori	33		
Josh Hursey	Ilya Ivanov	34		
Julien Jaeger	Emmanuel Jeannot	35		
Sylvain Jeaugey	Jithin Jose	36		
Krishna Kandalla	Takahiro Kawashima	37		
Chulho Kim	Michael Knobloch	38		
Alice Koniges	Sameer Kumar	39		
Kim Kyunghun	Ignacio Laguna Peralta	40		
Stefan Lankes	Tonglin Li	41		
Xioyi Lu	Kavitha Madhu	42		
Alexey Malhanov	Ryan Marshall	43		
William Marts	Guillaume Mercier	44		
Ali Mohammed	Kathryn Mohror	45		
		46		
		47		

_					
1 2	Takeshi Nanri Takafumi Nose	Thomas Naughton	Christoph Niethammer		
		Lena Oden	Steve Oyanagi Antonio Peña		
3	Guillaume Papauré Simon Pickartz	Ivy Peng Antere Delvelser	Sreeram Potluri		
4		Artem Polyakov	Marc Pérache		
5	Howard Pritchard	Martina Prugger			
6	Rolf Rabenseifner	Nicholas Radcliffe	Ken Raffenetti		
7	Craig Rasmussen	Soren Rasmussen	Hubert Ritzdorf		
8	Sergio Rivas-Gomez	Davide Rossetti	Martin Ruefenacht		
9	Amit Ruhela	Whit Schonbein	Joseph Schuchart		
10	Martin Schulz	Sangmin Seo	Sameh Sharkawi		
11	Sameer Shende	Min Si	Anthony Skjellum		
12	Brian Smith	David Solt	Jeffrey M. Squyres		
13	Srinivas Sridharan	Hari Subramoni	Nawrin Sultana		
14	Shinji Sumimoto	Sayantan Sur	Hugo Taboada		
15	Keita Teranishi	Rajeev Thakur	Keith Underwood		
16	Geoffroy Vallee	Akshay Venkatesh	Jerome Vienne		
17	Anh Vo	Justin Wozniak	Junchao Zhang		
18	Dong Zhong	Hui Zhou			
19					
20	The MPI Forum also address	lodges and approxist	es the valuable input from people via		
21	e-mail and in person.	ledges and appreciate	es the valuable input from people via		
22	-	ported the MPL 4.0 o	fort through time and travel support		
23	for the people listed above.		more infough time and travel support		
24					
25	ATOS				
26	Argonne National Laborator	У			
27	Arm				
28	Auburn University				
29	Barcelona Supercomputing C	Center			
30	CEA				
31	Cisco Systems Inc.				
32	Cray Inc.				
33	EPCC, The University of Ed	linburgh			
34	ETH Zürich				
35	Fujitsu				
36	Fulda University of Applied				
37	German Research School for	Simulation Sciences			
38	Hewlett Packard Enterprise				
39	International Business Mach				
40	Institut National de Recherc	he en Informatique e	t Automatique (Inria)		
41	Intel Corporation				
42	Jülich Supercomputing Cent	,	m Jülich		
43	KTH Royal Institute of Technology				
44	Kyushu University				
45	Lawrence Berkeley National Laboratory				
46	Lawrence Livermore National Laboratory				
47	Lenovo				
48	Los Alamos National Labora	tory			

Mellanox Technologies, Inc.	1
Microsoft Corporation	2
NEC Corporation	3
NVIDIA Corporation	4
Oak Ridge National Laboratory	5
PAR-TEC	6
Paratools, Inc.	7
RIKEN AICS (R-CCS as of 2017)	8
RWTH Aachen University	9
Rutgers University	10
Sandia National Laboratories	11
Silicon Graphics, Inc.	12
Technical University of Munich	13
The HDF Group	14
The Ohio State University	15
Texas Advanced Computing Center	16
Tokyo Institute of Technology	17
University of Alabama at Birmingham	18
University of Basel, Switzerland	19
University of Houston	20
University of Illinois at Urbana-Champaign and the National Center for Supercomput-	21
ing Applications	22
University of Innsbruck	23
University of Oregon	24
University of Potsdam	25
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)	26
University of Tennessee, Chattanooga	27
University of Tennessee, Knoxville	28
University of Texas at El Paso	29
University of Tokyo	30
VSC Research Center, TU Wien	31
	32

## Chapter 1

# Introduction to MPI

#### 1.1 Overview and Goals

MPI (Message-Passing Interface) is a *message-passing library interface specification*. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a *specification*, not an implementation; there are multiple implementations of MPI. This specification is for a *library interface*; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI's development.

 $^{24}$ 

 $^{31}$ 

 $45 \\ 46$ 

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of a messagepassing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases for which they can provide hardware support, thereby enhancing scalability.

The goal of the Message-Passing Interface, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processors, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran bindings for the interface.

- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

#### 1.2 Background of MPI-1.0

<sup>12</sup> MPI sought to make use of the most attractive features of a number of existing message-<sup>13</sup> passing systems, rather than selecting one of them and adopting it as the standard. Thus, <sup>14</sup> MPI was strongly influenced by work at the IBM T. J. Watson Research Center [2, 3], Intel's <sup>15</sup> NX/2 [57], Express [14], nCUBE's Vertex [53], p4 [9, 10], and PARMACS [6, 11]. Other <sup>16</sup> important contributions have come from Zipcode [60, 61], Chimp [20, 21], PVM [5, 18], <sup>17</sup> Chameleon [31], and PICL [26].

18 The MPI standardization effort involved about 60 people from 40 organizations mainly 19from the United States and Europe. Most of the major vendors of concurrent computers 20were involved in MPI, along with researchers from universities, government laboratories, and 21industry. The standardization process began with the Workshop on Standards for Message-22Passing in a Distributed Memory Environment, sponsored by the Center for Research on 23Parallel Computing, held April 29–30, 1992, in Williamsburg, Virginia [69]. At this work- $^{24}$ shop the basic features essential to a standard message-passing interface were discussed, 25and a working group established to continue the standardization process. 26

A preliminary draft proposal, known as MPI-1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [19]. MPI-1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI-1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI-1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at 34which it was decided to place the standardization process on a more formal footing, and to 35 generally adopt the procedures and organization of the High Performance Fortran Forum. 36 Subcommittees were formed for the major component areas of the standard, and an email 37 discussion service established for each. In addition, the goal of producing a draft MPI 38 standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 39 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI 40 standard at the Supercomputing 93 conference in November 1993. These meetings and the 41 email discussion together constituted the MPI Forum, membership of which has been open 42to all members of the high performance computing community. 43

44

 $45 \\ 46$ 

1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [23]. The first product of these deliberations

1

2

3

4

5 6

> 7 8

9 10

was Version 1.1 of the MPI specification, released in June of 1995 [24] (see http://www.mpi-forum.org for official MPI document releases). At that time, effort focused in five areas.

- 1. Further corrections and clarifications for the MPI-1.1 document.
- 2. Additions to MPI-1.1 that do not significantly change its types of functionality (new datatype constructors, language interoperability, etc.).
- 3. Completely new types of functionality (dynamic processes, one-sided communication, parallel I/O, etc.) that are what everyone thinks of as "MPI-2 functionality."
- 4. Bindings for Fortran 90 and C++. MPI-2 specifies C++ bindings for both MPI-1 and MPI-2 functions, and extensions to the Fortran 77 binding of MPI-1 and MPI-2 to handle Fortran 90 issues.
- 5. Discussions of areas in which the MPI process and framework seem likely to be useful, but where more discussion and experience are needed before standardization (e.g., zero-copy semantics on shared-memory machines, real-time specifications).

Corrections and clarifications (items of type 1 in the above list) were collected in Chapter 3 of the MPI-2 document: "Version 1.2 of MPI." That chapter also contains the function for identifying the version number. Additions to MPI-1.1 (items of types 2, 3, and 4 in the above list) are in the remaining chapters of the MPI-2 document, and constitute the specification for MPI-2. Items of type 5 in the above list have been moved to a separate document, the "MPI Journal of Development" (JOD), and are not part of the MPI-2 Standard.

This structure makes it easy for users and implementors to understand what level of MPI compliance a given implementation has:

- MPI-1 compliance will mean compliance with MPI-1.3. This is a useful level of compliance. It means that the implementation conforms to the clarifications of MPI-1.1 function behavior given in Chapter 3 of the MPI-2 document. Some implementations may require changes to be MPI-1 compliant.
- MPI-2 compliance will mean compliance with all of MPI-2.1.
- The MPI Journal of Development is not part of the MPI Standard.

It is to be emphasized that forward compatibility is preserved. That is, a valid MPI-1.1 program is both a valid MPI-1.3 program and a valid MPI-2.1 program, and a valid MPI-1.3 program is a valid MPI-2.1 program.

#### Background of MPI-1.3 and MPI-2.1 1.4

42After the release of MPI-2.0, the MPI Forum kept working on errata and clarifications for both standard documents (MPI-1.1 and MPI-2.0). The short document "Errata for MPI-1.1" was released October 12, 1998. On July 5, 2001, a first ballot of errata and clarifications for 44MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done electronically. Both ballots were combined into one document: "Errata for MPI-2," May 15, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors kept working on new requests for clarification.

1 2

3

4

56

7

8 9

10 11

12

13

1415

16

17

18

19

2021

22

23

 $^{24}$ 25

26

27

28

29

30

31 32

33

34 35

36

37

38 39

40

41

43

45

46

1 Restarting regular work of the MPI Forum was initiated in three meetings, at Eu- $\mathbf{2}$ roPVM/MPI'06 in Bonn, at EuroPVM/MPI'07 in Paris, and at SC'07 in Reno. In De-3 cember 2007, a steering committee started the organization of new MPI Forum meetings at 4 regular 8-weeks intervals. At the January 14–16, 2008 meeting in Chicago, the MPI Forum  $\mathbf{5}$ decided to combine the existing and future MPI documents to one document for each ver-6 sion of the MPI standard. For technical and historical reasons, this series was started with  $\overline{7}$ MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 8 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for 9 MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1–4) were combined into one draft document, 10 for each chapter, a chapter author and review team were defined. They cleaned up the 11document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document 12was finished in June 2008, and finally released with a second vote in September 2008 in the 13meeting at Dublin, just before EuroPVM/MPI'08.

 $14 \\ 15$ 

16 17

18

19

20 21

22

23 24

### 1.5 Background of MPI-2.2

MPI-2.2 is a minor update to the MPI-2.1 standard. This version addresses additional errors and ambiguities that were not corrected in the MPI-2.1 standard as well as a small number of extensions to MPI-2.1 that met the following criteria:

- Any correct MPI-2.1 program is a correct MPI-2.2 program.
- Any extension must have significant benefit for users.
- Any extension must not require significant implementation effort. To that end, all such changes are accompanied by an open source implementation.

The discussions of MPI-2.2 proceeded concurrently with the MPI-3 discussions; in some cases, extensions were proposed for MPI-2.2 but were later moved to MPI-3.

### 1.6 Background of MPI-3.0

MPI-3.0 is a major update to the MPI standard. The updates include the extension of collective operations to include nonblocking versions, extensions to the one-sided operations, and a new Fortran 2008 binding. In addition, the deprecated C++ bindings have been removed, as well as many of the deprecated routines and MPI objects (such as the MPI\_UB datatype).

### 1.7 Background of MPI-3.1

<sup>40</sup> <sup>41</sup> MPI-3.1 is a minor update to the MPI standard. Most of the updates are corrections <sup>42</sup> and clarifications to the standard, especially for the Fortran bindings. New functions added <sup>43</sup> include routines to manipulate MPI\_Aint values in a portable manner, nonblocking collective <sup>44</sup> I/O routines, and routines to get the index value by name for MPI\_T performance and <sup>45</sup> control variables. A general index was also added.

- 46
- 47
- 48

25 26 27

28

29 30

31 32

33

34

35

36

37 38

#### 1.8 Background of MPI-4.0

MPI-4.0 is a major update to the MPI standard. The largest changes are the addition of large-count versions of many routines to address the limitations of using an int or INTEGER for the count parameter, persistent collectives, partitioned communications, an alternative way to initialize MPI, application info assertions, and improvements to the definitions of error handling. In addition, there are a number of smaller improvements and corrections.

#### 1.9 Who Should Use This Standard?

This standard is intended for use by all those who want to write portable message-passing programs in Fortran and C (and access the C bindings from C++). This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

#### 1.10 What Platforms Are Targets for Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the sharedand distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those "machines" consisting of collections of other machines, parallel or not, connected by a communication network.

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogenous networks of workstations.

#### 1.11 What Is Included in the Standard?

The standard includes:

- Point-to-point communication,
- Partitioned communication,
- Datatypes,
- Collective operations,

1 2

3

4

5

6

7 8 9

10 11

12

13

14

15

20 21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41 42

43 44

 $45 \\ 46$ 

47

• Process topologies,

- The Info object,
  - Process initialization, creation, and management,
- One-sided communication,
- External interfaces.
- Parallel file I/O,
  - Tool support,
    - Language bindings for Fortran and C.

#### What Is Not Included in the Standard? 1.12

The standard does not specify:

- Operations that require more operating system support than is currently standard; for example, interrupt-driven receives, remote execution, or active messages,
- Program construction tools,
- Debugging facilities.

There are many features that have been considered and not included in this standard. This happened for a number of reasons, one of which is the time constraint that was selfimposed in finishing the standard. Features that are not included can always be offered as extensions by specific implementations. Perhaps future versions of MPI will address some of these issues.

1.13Organization of This Document

The following is a list of the remaining chapters in this document, along with a brief description of each.

- Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
- Chapter 3, Point-to-Point Communication, defines the basic, pairwise communication subset of MPI. Send and receive are found here, along with many associated functions designed to make basic communication powerful and efficient.

6

1

 $\mathbf{2}$ 

3 4

56

7

8 9

10 11

12

13 14

1516

17

18 1920

2122

23

 $^{24}$ 

2526

2728

2930

 $^{31}$ 

32

33

34

35 36

37 38

39

40 41

42

43

44

4546

- Chapter 4, Partitioned Point-to-Point Communication, defines a method of performing partitioned communication in MPI. Partitioned communication allows multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single message.
- Chapter 5, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.
- Chapter 6, Collective Communication, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include inter-communicators. It also adds two new collective operations. MPI-3 adds nonblocking collective operations. MPI-4 adds persistent nonblocking collective operations.
- Chapter 7, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a *communicator*.
- Chapter 8, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 9, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- Chapter 10, The Info Object, defines an opaque object, that is used as input in several MPI routines.
- Chapter 11, Process Initialization, Creation, and Management, defines several approaches to MPI initialization, process creation, and process management while placing minimal restrictions on the execution environment. MPI-4 adds a new Sessions Model.
- Chapter 12, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 13, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 14, I/O, defines MPI support for parallel I/O.
- Chapter 15, Tool Support, covers interfaces that allow debuggers, performance analyzers, and other tools to obtain data about the operation of MPI processes. This chapter includes Section 15.2 (Profiling Interface), which was a chapter in previous versions of MPI.

 $^{24}$ 

 $^{31}$ 

1 2 3	• Chapter 16, Deprecated Interfaces, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.
4 5 6 7	• Chapter 17, Removed Interfaces, describes routines and constructs that have been removed from MPI. These were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard.
8 9 10	• Chapter 18, Semantic Changes and Warnings, describes semantic changes from pre- vious versions of MPI.
11 12	• Chapter 19, Language Bindings, discusses Fortran issues, and describes language in- teroperability aspects between C and Fortran.
13 14	The Appendices are:
15 16 17	• Annex A, Language Bindings Summary, gives specific syntax in C and Fortran, for all MPI functions, constants, and types.
18 19 20	• Annex B, Change-Log, summarizes some changes since the previous version of the standard.
21 22 23	• Several Index pages show the locations of general terms and definitions, examples, con- stants and predefined handles, declarations of C and Fortran types, callback routine prototypes, and all MPI functions.
24 25 26 27 28 29 30 31 32	MPI provides various interfaces to facilitate interoperability of distinct MPI imple- mentations. Among these are the canonical data representation for MPI I/O and for MPI_PACK_EXTERNAL and MPI_UNPACK_EXTERNAL. The definition of an actual bind- ing of these interfaces that will enable interoperability is outside the scope of this document. A separate document consists of ideas that were discussed in the MPI Forum during the MPI-2 development and deemed to have value, but are not included in the MPI Standard. They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order to provide a starting point for further work. The chapters in the JOD are
33 34 35 36 37	• Chapter 2, Spawning Independent Processes, includes some elements of dynamic processes management, in particular management of processes with which the spawning processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard.
38 39 40	• Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multithreaded environment.
40 41 42	• Chapter 4, Communicator ID, describes an approach to providing identifiers for com- municators.
43 44 45 46	• Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.
47 48	• Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.

Chapter 8	Real-Time MPI, discusses MPI support for real time processing.	
chapter o,		

## Chapter 2

# **MPI** Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices.

 $^{31}$ 

#### 2.1 Document Notation

*Rationale.* Throughout this document, the rationale for the design choices made in the interface specification is set off in this format. Some readers may wish to skip these sections, while readers interested in interface design may want to read them carefully. (*End of rationale.*)

Advice to users. Throughout this document, material aimed at users and that illustrates usage is set off in this format. Some readers may wish to skip these sections, while readers interested in programming in MPI may want to read them carefully. (*End of advice to users.*)

Advice to implementors. Throughout this document, material that is primarily commentary to implementors is set off in this format. Some readers may wish to skip these sections, while readers interested in MPI implementations may want to read them carefully. (*End of advice to implementors.*)

#### 2.2 Naming Conventions

In many cases MPI names for C functions are of the form MPI\_Class\_action\_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules.

- In C and the Fortran mpi\_f08 module, all routines associated with a particular type of MPI object should be of the form MPI\_Class\_action\_subset or, if no subset exists, of the form MPI\_Class\_action. In the Fortran mpi module and mpif.h file, all routines associated with a particular type of MPI object should be of the form MPI\_CLASS\_ACTION\_SUBSET or, if no subset exists, of the form MPI\_CLASS\_ACTION.
- 2. If the routine is not associated with a class, the name should be of the form MPI\_Action\_subset or MPI\_ACTION\_SUBSET in C and Fortran.

3. The names of certain actions have been standardized. In particular, **Create** creates a new object, **Get** retrieves information about an object, **Set** sets this information, **Delete** deletes information, **Is** asks whether or not an object has a certain property.

C and Fortran names for some MPI functions (that were defined during the MPI-1 process) violate these rules in several cases. The most common exceptions are the omission of the **Class** name from the routine and the omission of the **Action** where one can be inferred.

#### 2.3 Procedure Specification

MPI procedures are specified using a language-independent notation. The arguments of procedure calls are marked as IN, OUT, or INOUT. The meanings of these are:

- IN: the call may use the input value but does not update the argument from the perspective of the caller at any time during the call's execution,
- 16 17 18

19

26

27

28 29

- OUT: the call may update the argument but does not use its input value,
- INOUT: the call may both use and update the argument.

There is one special case—if an argument is a handle to an opaque object (these terms are defined in Section 2.5.1), and the object is updated by the procedure call, then the argument is marked INOUT or OUT. It is marked this way even though the handle itself is not modified—we use the INOUT or OUT attribute to denote that what the handle *references* is updated.

*Rationale.* The definition of MPI tries to avoid, to the largest possible extent, the use of INOUT arguments, because such use is error-prone, especially for scalar arguments. (*End of rationale.*)

MPI's use of IN, OUT, and INOUT is intended to indicate to the user how an argument is to be used, but does not provide a rigorous classification that can be translated directly into all language bindings (e.g., INTENT in Fortran 90 bindings or const in C bindings). For instance, the "constant" MPI\_BOTTOM can usually be passed to OUT buffer arguments. Similarly, MPI\_STATUS\_IGNORE can be passed as the OUT status argument.

A common occurrence for MPI functions is an argument that is used as IN by some processes and OUT by other processes. Such an argument is, syntactically, an INOUT argument and is marked as such, although, semantically, it is not used in one call both for input and for output on a single process.

Another frequent situation arises when an argument value is needed only by a subset of the processes. When an argument is not significant at a process then an arbitrary value can be passed as an argument.

<sup>42</sup> Unless specified otherwise, an argument of type OUT or type INOUT cannot be aliased <sup>43</sup> with any other argument passed to an MPI procedure. An example of argument aliasing in <sup>44</sup> C appears below. If we define a C procedure like this,

```
45 void copyIntBuffer(int *pin, int *pout, int len)
46 { int i;
47 for (i=0; i<len; ++i) *pout++ = *pin++;
48 }</pre>
```

1

2

3

4

5

6

7

8 9 10

11 12

13 14

then a call to it in the following code fragment has aliased arguments.

int a[10]; copyIntBuffer(a, a+3, 7);

Although the C language allows this, such usage of MPI procedures is forbidden unless otherwise specified. Note that Fortran prohibits aliasing of arguments.

All MPI functions are first specified in the language-independent notation. Immediately below this, language dependent bindings follow:

- The ISO C version(s) of the function.
- The Fortran version(s) used with USE mpi\_f08.
- The Fortran version of the same function used with USE mpi or INCLUDE 'mpif.h'.

Some MPI procedures have two interfaces for a given language support; see Sections 2.5.6 and 2.5.8.

An exception is Section 15.3 "The MPI Tool Information Interface", which only provides ISO C interfaces.

"Fortran" in this document refers to Fortran 90 and higher; see Section 2.6.

The words function, routine, procedure, procedure call, and call are often used as synonyms within this standard.

#### 2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used. The term **message** data buffer refers to the send/receive buffer used in a communication procedure. The term file data buffer refers to the data buffers used by MPI I/O procedures. In this section we use the term data buffer and depending on the MPI procedure it will refer to message data buffer or file data buffer.

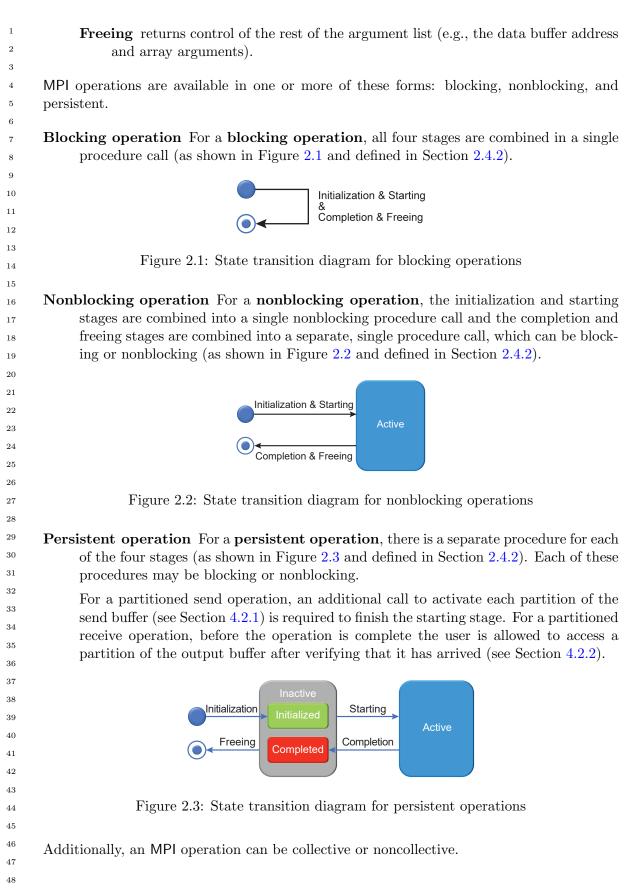
#### 2.4.1 MPI Operations

- **MPI operation** An MPI operation is a sequence of steps performed by the MPI library to establish and enable data transfer and/or synchronization. It consists of four stages: initialization, starting, completion, and freeing, and it is implemented as a set of one or more MPI procedures, see Section 2.4.2.
  - **Initialization** hands over the argument list to the operation but not the content of the data buffers, if any. The specification of an operation may state that array arguments must not be changed until the operation is freed.
  - **Starting** hands over the control of the data buffers, if any, to the associated operation.

Note that **initiation** refers to the combination of the initialization and starting stages.

**Completion** returns control of the content of the data buffers and indicates that output buffers and arguments, if any, have been updated.

Note that an MPI operation is **complete** when the MPI procedure implementing the completion stage returns.



 $45 \\ 46$ 

47 48

<b>Collective operation</b> Collective operations are defined as operations that involve a group or groups of MPI processes. For collective operations the completion stage may or	1 2		
may not finish before all processes in the group have started the operation.	3		
Collective MPI operations are also available as blocking, nonblocking, or persistent operations.	4		
	6		
<b>Noncollective operation</b> Noncollective operations are defined as operations that are not collective.	7 8		
	9		
2.4.2 MPI Procedures	10 11		
All MPI procedures can either be <i>local</i> or <i>non-local</i> —defined as follows:	11		
Non-local procedure An MPI procedure is non-local if returning may require, during its	13		
execution, some specific semantically-related MPI procedure to be called on another MPI process.	14 15		
	16		
Local procedure An MPI procedure is local if it is not <i>non-local</i> .	17		
An MPI operation is implemented as a set of one or more MPI procedures. An MPI	18		
operation-related procedure implements at least a part of a stage of an MPI operation	19		
as described in Section 2.4.1. An MPI operation-related procedure may also implement	20		
one or more stages of one or several MPI operations. In certain cases, more than one MPI	21 22		
operation-related procedure may be needed to implement a single stage.	22		
There are also other MPI procedures that do not implement any stage of any MPI	23 24		
operation.	25		
The semantics of MPI operation-related procedures are described using two orthogonal (independent) concepts: completeness (depends on which stages are included) and locality.	26 27		
Such procedures can be either incomplete, or completing, or freeing, or completing and	28		
freeing based on the status of the associated operation at the time the procedure returns.	29		
Also, all such procedures can be described as either blocking or nonblocking, but these latter	30		
two terms refer to combinations of the completeness and locality concepts. Additionally,	31		
all MPI operation-related procedures can be collective or noncollective.	32		
The following are properties of MPI operation-related procedures:	33		
Initialization procedure An MPI procedure is an initialization procedure if return	34		
from the procedure indicates that the associated operation has completed its initial-	35		
ization stage, which implies that the user has handed over control of the argument list	36		
(but not contents of the data buffers) to MPI. The user is still allowed to read or mod-	37		
ify the contents of the data buffers. If an initializing procedure is not also the freeing	38 39		
procedure of the associated operation (see below) then the user is not permitted to			
deallocate the data buffers or to modify the array arguments.	40 41		
Starting procedure An MPI procedure is a starting procedure if return from the pro-	42		
cedure indicates that the associated operation has completed its starting stage, which	43		
implies that the user has handed over control of the data buffers to MPI. If a starting	44		

cedure indicates that the associated operation has completed its starting stage, which implies that the user has handed over control of the data buffers to MPI. If a starting procedure is not also a completing procedure of the associated operation (see below) then the user is not permitted to modify input data buffers or to read output data buffers.

1 2 3	<b>Initiation procedure</b> An MPI procedure is an <b>initiation procedure</b> if return from the procedure indicates that both the initialization and the starting stage have completed, which implies control of the entire argument list is handed over to MPI.
4 5 7 8 9 10	<b>Completing procedure</b> An MPI procedure is called <b>completing</b> if return from the procedure indicates that at least one associated operation has finished its completion stage, which implies that the user can rely on the content of the output data buffers and modify the content of input and output data buffers of such operation(s). If a completing procedure is not also a freeing procedure (see below) then the user is not permitted to deallocate the data buffers or to modify the array arguments.
11 12 13	<b>Incomplete procedure</b> An MPI procedure is called <b>incomplete</b> if it is not a completing procedure.
14 15 16 17 18	<b>Freeing procedure</b> An MPI procedure is <b>freeing</b> if return from the procedure indicates that at least one associated operation has finished its freeing stage, which implies that the user can reuse all parameters specified when initializing such associated operation(s).
19	Nonblocking procedure An MPI procedure is nonblocking if it is incomplete and local.
20 21	Blocking procedure An MPI procedure is blocking if it is not nonblocking.
22 23 24 25 26	Advice to users. Note that for operation-related MPI procedures, in most cases incomplete procedures are local and completing procedures are non-local. Exceptions are noted where such procedures are defined. In many cases an additional prefix letter I as an abbreviation of the words <b>incomplete</b> and <b>immediate</b> marks nonblocking procedures in the procedure name.
27	Some categorization examples are listed below.
28 29	Nonblocking procedures:
30 31 32	• incomplete and local: MPI_ISEND, MPI_IRECV, MPI_IBCAST, MPI_IMPROBE, MPI_SEND_INIT, MPI_RECV_INIT,
33	Blocking procedures:
34 35 36 37 38	<ul> <li>completing and non-local: MPI_SEND, MPI_RECV, MPI_BCAST,</li> <li>incomplete and non-local: MPI_MPROBE, MPI_BCAST_INIT,, MPI_FILE_{READ WRITE}_{AT_ALL ALL ORDERED}_BEGIN.</li> <li>completing and local: MPI_BSEND, MPI_RSEND, MPI_MRECV.</li> </ul>
39 40	MPI procedures that are not MPI operation-related:
41	• MPI_COMM_RANK, MPI_WTIME, MPI_PROBE, MPI_IPROBE,
42 43	(End of advice to users.)
44 45 46	<b>Collective procedure</b> An MPI procedure is <b>collective</b> if all processes in a group or groups of MPI processes need to invoke the procedure.
47 48	Initialization procedures of collective operations over the same process group must be executed in the same order by all members of the process group.

An MPI collective procedure is **synchronizing** if it will only return once all processes in the associated group or groups of MPI processes have called the appropriate matching MPI procedure.

The initiation procedures for nonblocking collective operations and the starting procedures for persistent collective operations are local and shall not be synchronizing.

All other procedures for collective operations, such as for blocking collective operations and the initialization procedures for persistent collective operations, may or may not be synchronizing.

Advice to users. Calling any synchronizing function is erroneous when there is no possibility of corresponding calls at all other processes in the associated process group.

Waiting for completion of any collective operation is erroneous when there is no possibility that all other processes in the associated group will be able to start the corresponding operation. (*End of advice to users.*)

#### 2.4.3 MPI Datatypes

For datatypes, the following terms are defined:

predefined A predefined datatype is a datatype with a predefined (constant) name (such as MPI\_INT, MPI\_FLOAT\_INT, or MPI\_PACKED) or a datatype constructed with MPI\_TYPE\_CREATE\_F90\_INTEGER, MPI\_TYPE\_CREATE\_F90\_REAL, or MPI\_TYPE\_CREATE\_F90\_COMPLEX. The former are named whereas the latter are unnamed.

derived A derived datatype is any datatype that is not predefined.

- **portable** A datatype is portable if it is a predefined datatype, or it is derived from 28 a portable datatype using only the type constructors MPI\_TYPE\_CONTIGUOUS, 29MPI\_TYPE\_VECTOR, MPI\_TYPE\_INDEXED, 30 MPI\_TYPE\_CREATE\_INDEXED\_BLOCK, MPI\_TYPE\_CREATE\_SUBARRAY,  $^{31}$ MPI\_TYPE\_DUP, and MPI\_TYPE\_CREATE\_DARRAY. Such a datatype is portable 32 because all displacements in the datatype are in terms of extents of one predefined 33 datatype. Therefore, if such a datatype fits a data layout in one memory, it will 34 fit the corresponding data layout in another memory, if the same declarations were 35 used, even if the two systems have different architectures. On the other hand, if a 36 datatype was constructed using MPI\_TYPE\_CREATE\_HINDEXED, 37 MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK, MPI\_TYPE\_CREATE\_HVECTOR or 38 MPI\_TYPE\_CREATE\_STRUCT, then the datatype contains explicit byte displace-39 ments (e.g., providing padding to meet alignment restrictions). These displacements 40 are unlikely to be chosen correctly if they fit data layout on one memory, but are 41 used for data layouts on another process, running on a processor with a different 42architecture. 43 44
- **equivalent** Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

1 2

3

4

5

6

 $\overline{7}$ 

8

9 10

11

12

13

14

15 16

17 18

19 20

21

22

23

24

25 26

27

45

46

### 2.5 Datatypes

#### 2.5.1 Opaque Objects

MPI manages **system memory** that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc. This memory is not directly accessible to the user, and objects stored there are **opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. MPI procedures that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

In Fortran with USE mpi or INCLUDE 'mpif.h', all handles have type INTEGER. In Fortran with USE mpi\_f08, and in C, a different handle type is defined for each category of objects. With Fortran USE mpi\_f08, the handles are defined as Fortran BIND(C) derived types that consist of only one element INTEGER :: MPI\_VAL. The internal handle value is identical to the Fortran INTEGER value used in the mpi module and mpif.h. The operators .EQ., .NE., == and /= are overloaded to allow the comparison of these handles. The type names are identical to the names in C, except that they are not case sensitive. For example:

```
<sup>19</sup> TYPE, BIND(C) :: MPI_Comm
```

INTEGER :: MPI\_VAL

END TYPE MPI\_Comm

The C types must support the use of the assignment and equality operators.

Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. (End of advice to implementors.)

Rationale. Since the Fortran integer values are equivalent, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle COMM can be converted directly into an exactly equivalent mpi\_f08 communicator handle named comm\_f08 by comm\_f08%MPI\_VAL=COMM, and vice versa. The use of the INTEGER defined handles and the BIND(C) derived type handles is different: Fortran 2003 (and later) define that BIND(C) derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage units are used for these BIND(C) derived type handles. Most compilers use one unit for both, the INTEGER handles and the handles defined as BIND(C) derived types. (End of rationale.)

Advice to users. If a user wants to substitute mpif.h or the mpi module by the mpi\_f08 module and the application program stores a handle in a Fortran common block then it is necessary to change the Fortran support method in all application routines that use this common block, because the number of numerical storage units of such a handle can be different in the two modules. (End of advice to users.)

44

<sup>45</sup> Opaque objects are allocated and deallocated by calls that are specific to each object <sup>46</sup> type. These are listed in the sections where the objects are described. The calls accept a <sup>47</sup> handle argument of matching type. In an allocate call this is an OUT argument that returns <sup>48</sup> a valid reference to the object. In a call to deallocate this is an INOUT argument which

1 2

3 4

5

6

7

8

9

10

20

21

22

23 24

25

26

27 28

29

30

31

32

33

34

35

36

37

38 39

40

41

42

1 2

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22

23 24

25

26

27 28

29

30

31

32

33

34

35

36

37

38

39

returns with an "invalid handle" value. MPI provides an "invalid handle" constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to a deallocate routine invalidates the handle and marks the object for deallocation. The object is not accessible to the user after the call. However, MPI need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

An opaque object and its handle are significant only at the process where the object was created and cannot be transferred to another process.

MPI provides certain predefined opaque objects and predefined, static handles to these objects. The user must not free such objects.

*Rationale.* This design hides the internal representation used for MPI data structures, thus allowing similar calls in C and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows future extensions of functionality. The mechanism for opaque objects used here loosely follows the POSIX Fortran binding standard.

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative in C would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. In Fortran, the handles are defined such that assignment and comparison are available through the operators of the language or overloaded versions of these operators. (*End of rationale.*)

Advice to users. A user may accidentally create a dangling reference by assigning to a handle the value of another handle, and then deallocating the object associated with these handles. Conversely, if a handle variable is deallocated before the associated object is freed, then the object becomes inaccessible (this may occur, for example, if the handle is a local variable within a subroutine, and the subroutine is exited before the associated object is deallocated). It is the user's responsibility to avoid adding or deleting references to opaque objects, except as a result of MPI calls that allocate or deallocate such objects. (*End of advice to users.*)

Advice to implementors. The intended semantics of opaque objects is that opaque 40 41 objects are separate from one another; each call to allocate such an object copies 42all the information required for the object. Implementations may avoid excessive 43copying by substituting referencing for copying. For example, a derived datatype 44may contain references to its components, rather than copies of its components; a call to MPI\_COMM\_GROUP may return a reference to the group associated with the 4546communicator, rather than a copy of this group. In such cases, the implementation 47must maintain reference counts, and allocate and deallocate objects in such a way that 48 the visible effect is as if the objects were copied. (End of advice to implementors.)

#### 2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this 6 number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the size of the entire array. The same approach is followed for other array arguments. In some cases NULL handles are considered valid entries. When a NULL argument is desired for an array of statuses, one 10 uses MPI\_STATUSES\_IGNORE.

11 12

1

2

3

4

5

7

8

9

#### 2.5.3 State 13

14MPI procedures use at various places arguments with *state* types. The values of such a 15datatype are all identified by names, and no operation is defined on them. For example, 16the MPI\_TYPE\_CREATE\_SUBARRAY routine has a state argument order with values 17MPI\_ORDER\_C and MPI\_ORDER\_FORTRAN.

18 19

20

29

#### 2.5.4 Named Constants

21MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, 22 with a special wild-card value, MPI\_ANY\_TAG. Such arguments will have a range of regular 23values, which is a proper subrange of the range of values of the corresponding basic type;  $^{24}$ special values (such as MPI\_ANY\_TAG) will be outside the regular range. The range of regu-2526lar values, such as tag, can be queried using environmental inquiry functions, see Chapter 9. The range of other values, such as source, depends on values given by other MPI routines 27(in the case of source it is the communicator size). 28

MPI also provides predefined named constant handles, such as MPI\_COMM\_WORLD.

All named constants, with the exceptions noted below for Fortran, can be used in 30 initialization expressions or assignments, but not necessarily in array declarations or as  $^{31}$ labels in C switch or Fortran select/case statements. This implies named constants 32 to be link-time but not necessarily compile-time constants. The named constants listed 33 34below are required to be compile-time constants in both C and Fortran. These constants do not change values during execution. Opaque objects accessed by constant handles are 35 defined and do not change value between MPI initialization (MPI\_INIT) and MPI completion 36 (MPI\_FINALIZE). The handles themselves are constants and can be also used in initialization 37 expressions or assignments. 38

39 The constants that are required to be compile-time constants (and can thus be used for array length declarations and labels in C switch and Fortran case/select statements) 4041 are:

- 42MPI\_MAX\_PROCESSOR\_NAME
- MPI\_MAX\_LIBRARY\_VERSION\_STRING 43
- MPI\_MAX\_ERROR\_STRING 44
- MPI\_MAX\_DATAREP\_STRING 45
- MPI\_MAX\_INFO\_KEY 46
- 47 MPI\_MAX\_INFO\_VAL
- MPI\_MAX\_OBJECT\_NAME 48

MPI_MAX_PORT_NAME	1
MPI_VERSION	2
MPI_SUBVERSION	3
MPI_F_STATUS_SIZE (C only)	4
MPI_STATUS_SIZE (Fortran only)	5
MPI_ADDRESS_KIND (Fortran only)	6
MPI_COUNT_KIND (Fortran only)	7
MPI_INTEGER_KIND (Fortran only)	8
MPI_OFFSET_KIND (Fortran only)	9
MPI_SUBARRAYS_SUPPORTED (Fortran only)	10
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)	11
The constants that cannot be used in initialization expressions or assignments in Fortran	12
are as follows:	13
MPI_BOTTOM	14
MPI_STATUS_IGNORE	15
MPI_STATUSES_IGNORE	16
MPI_ERRCODES_IGNORE	17
MPI_IN_PLACE	18
MPI_ARGV_NULL	19
MPI_ARGVS_NULL	20
MPI_UNWEIGHTED	21
MPI_WEIGHTS_EMPTY	22

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through PARAMETER statements) is not possible because an implementation cannot distinguish these values from valid data. Typically, these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, this address can be extracted by some mechanism outside the Fortran standard (e.g., by Fortran extensions or by implementing the function in C). (End of advice to implementors.)

#### 2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran with the include file mpif.h or the mpi module, the document uses <type> to represent a choice variable; with the Fortran mpi\_f08 module, such arguments are declared with the Fortran 2008 + TS 29113 syntax TYPE(\*), DIMENSION(..); for C, we use void\*.

Advice to implementors. Implementors can freely choose how to implement choice arguments in the mpi module, e.g., with a nonstandard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the mpi\_f08 module. See details in Section 19.1.1. (End of advice to implementors.)

 $^{24}$ 

 $^{31}$ 

#### 2.5.6 Absolute Addresses and Relative Address Displacements

Some MPI procedures use *address* arguments that represent an *absolute address* in the calling program, or *relative displacement* arguments that represent differences of two absolute addresses. The datatype of such arguments is MPI\_Aint in C and INTEGER(KIND=

MPI\_ADDRESS\_KIND) in Fortran. These types must have the same width and encode address 6 values in the same manner such that address values in one language may be passed directly 7 to another language without conversion. There is the MPI constant MPI\_BOTTOM to in-8 dicate the start of the address range. For retrieving absolute addresses or any calculation 9 with absolute addresses, one should use the routines and functions provided in Section 5.1.5. 10 Section 5.1.12 provides additional rules for the correct use of absolute addresses. For ex-11 pressions with relative displacements or other usage without absolute addresses, intrinsic 12operators (e.g., +, -, \*) can be used. 13

Rationale. Byte displacement values need to be large enough to encode any value used for expressing absolute or relative memory addresses. Prior to MPI-4.0, some MPI routines used int in C and INTEGER in Fortran as the type for byte displacement arguments. To avoid breaking backward compatibility, this version of the standard continues to support int in C as well as INTEGER in Fortran in such routines. In addition, this version of the standard supports using MPI\_Aint in C (via separate "\_c"suffixed procedures) as well as INTEGER(KIND=MPI\_ADDRESS\_KIND) in Fortran (via polymorphic interfaces in newer MPI Fortran bindings (USE mpi\_f08)) in such routines. See Section 19.2 for a full explanation. (End of rationale.)

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities
 can easily be larger than 32 bits which can be the default size of a Fortran integer. To
 overcome this, these quantities are declared to be INTEGER(KIND=MPI\_OFFSET\_KIND) in
 Fortran. In C one uses MPI\_Offset. These types must have the same width and encode
 address values in the same manner such that offset values in one language may be passed
 directly to another language without conversion.

2.5.8 Counts

As described above, MPI defines types (e.g., MPI\_Aint) to address locations within memory and other types (e.g., MPI\_Offset) to address locations within files. In addition, some MPI procedures use *count* arguments that represent a number of MPI datatypes on which to operate. Furthermore, *timestamps* in the context of the MPI Tool Information Interface are a count of clock ticks elapsed since some time in the past. At times, one needs a single type that can be used to address locations within either memory or files as well as express *count* values, and that type is MPI\_Count in C and

INTEGER(KIND=MPI\_COUNT\_KIND) in Fortran. These types must have the same width and encode values in the same manner such that count values in one language may be passed directly to another language without conversion. The size of the MPI\_Count type is determined by the MPI implementation with the restriction that it must be minimally capable of encoding any value that may be stored in a variable of type int, MPI\_Aint, or MPI\_Offset in C and of type INTEGER, INTEGER(KIND=MPI\_ADDRESS\_KIND), or INTEGER(KIND=MPI\_OFFSET\_KIND)

1

14

15

16

17

18

19

20

21

22

23 24

25

33

in Fortran. Even though the MPI\_Count type is large enough to encode address locations, the MPI\_Count type shall not be used to represent an *absolute address*.

Rationale. Count values need to be large enough to encode any value used for expressing element counts, strides, offsets, indexes, displacements, typemaps in memory, typemaps in file views, etc. Prior to MPI-4.0, many MPI routines used int in C and INTEGER in Fortran as the type for *count* arguments. To avoid breaking backward compatibility, this version of the standard continues to support int in C as well as INTEGER in Fortran in such routines. In addition, this version of the standard supports using MPI\_Count in C (via separate "\_c"suffixed procedures) as well as INTEGER(KIND=MPI\_COUNT\_KIND) in Fortran (via polymorphic interfaces in newer MPI Fortran bindings (USE mpi\_f08)) in such routines. See Section 19.2 for a full explanation. (*End of rationale.*)

The phrase **large count** refers to the use of MPI\_Count and INTEGER(KIND=MPI\_COUNT\_KIND) parameter types.

There are cases where MPI\_UNDEFINED can be returned in a **large count** OUT parameter. Per Table A.1.1 (page 859), the MPI\_UNDEFINED constant is defined to be a C int (or unnamed enum) and a Fortran INTEGER. Implementations shall therefore choose the underlying types for MPI\_Count and INTEGER(KIND=MPI\_COUNT\_KIND) such that they can be compared to MPI\_UNDEFINED.

Advice to implementors. The comparison of MPI\_UNDEFINED to an MPI\_Count or INTEGER(KIND=MPI\_COUNT\_KIND) may need to be via a casting operation. (End of advice to implementors.)

#### 2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, and ISO C, in particular. (Note that ANSI C has been replaced by ISO C.) Defined here are various object representations, as well as the naming conventions used for expressing this standard. The actual calling sequences are defined elsewhere.

MPI bindings are for Fortran 90 or later, though they were originally designed to be usable in Fortran 77 environments. With the mpi\_f08 module, two new Fortran features, assumed type (i.e., TYPE(\*)) and assumed rank (i.e., DIMENSION(..)), are also required, see Section 2.5.5.

Since the word **PARAMETER** is a keyword in the Fortran language, we use the word "argument" to denote the arguments to a subroutine. These are normally referred to as parameters in C, however, we expect that C programmers will understand the word "argument" (which has no specific meaning in C), thus allowing us to avoid unnecessary confusion for Fortran programmers.

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid any prefix of the form "MPI\_" and "PMPI\_", where any of the letters are either upper or lower case.

#### 2.6.1 Deprecated and Removed Interfaces

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 16, but that users

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

 $^{31}$ 

1 are recommended not to continue using, since better solutions were provided with newer  $\mathbf{2}$ versions of MPI. For example, the Fortran binding for MPI-1 functions that have address 3 arguments uses INTEGER. This is not consistent with the C binding, and causes problems on 4 machines with 32 bit INTEGERs and 64 bit addresses. In MPI-2, these functions were given  $\mathbf{5}$ new names with new bindings for the address arguments. The use of the old functions was 6 declared as deprecated. For consistency, here and in a few other cases, new C functions are 7also provided, even though the new functions are equivalent to the old functions. The old 8 names are deprecated.

<sup>9</sup> Some of the previously deprecated constructs are now removed, as documented in
 <sup>10</sup> Chapter 17. They may still be provided by an implementation for backwards compatibility,
 <sup>11</sup> but are not required.

Table 2.1 shows a list of all of the deprecated and removed constructs. Note that some
 C typedefs and Fortran subroutine names are included in this list; they are the types of
 callback functions.

15 16

17

2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term "Fortran" is used it means Fortran 90 or later; it means Fortran 2008 + TS 29113 and later if the mpi\_f08 module is used.

All MPI names have an MPI\_ prefix, and all characters are capitals. Programs must not declare names, e.g., for variables, subroutines, functions, parameters, derived types, abstract interfaces, or modules, beginning with the prefix MPI\_. To avoid conflicting with the profiling interface, programs must also avoid subroutines and functions with the prefix PMPI\_. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. With USE mpi\_f08, this last argument is declared as OPTIONAL, except for user-defined callback functions (e.g., COMM\_COPY\_ATTR\_FUNCTION) and their predefined callbacks (e.g., MPI\_COMM\_NULL\_COPY\_FN). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI\_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 9 and Annex A.

<sup>35</sup> Constants representing the maximum length of a string are one smaller in Fortran than <sup>36</sup> in C as discussed in Section 19.3.9.

Handles are represented in Fortran as INTEGERs, or as a BIND(C) derived type with the mpi\_f08 module; see Section 2.5.1. Binary-valued variables are of type LOGICAL.

39

37

38

Array arguments are indexed from one.

The older MPI Fortran bindings (mpif.h and use mpi) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 19.1.16.

The support for large count and displacement in Fortran is only available when using newer MPI Fortran bindings (USE mpi\_f08). For better readability, all Fortran large count procedure declarations are marked with a comment "!(\_c)".

- 46
- 47
- 48

#### 2.6. LANGUAGE BINDING

Deprecated or removed	deprecated	removed	Replacement
construct	since	since	
MPI_ADDRESS	MPI-2.0	MPI-3.0	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI-2.0	MPI-3.0	MPI_TYPE_GET_EXTENT
MPI_LB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
MPI_UB <sup>1</sup>	MPI-2.0	MPI-3.0	MPI_TYPE_CREATE_RESIZED
MPI_ERRHANDLER_CREATE	MPI-2.0	MPI-3.0	MPI_COMM_CREATE_ERRHANDLER
MPI_ERRHANDLER_GET	MPI-2.0	MPI-3.0	MPI_COMM_GET_ERRHANDLER
MPI_ERRHANDLER_SET	MPI-2.0	MPI-3.0	MPI_COMM_SET_ERRHANDLER
MPI_Handler_function <sup>2</sup>	MPI-2.0	MPI-3.0	MPI_Comm_errhandler_function <sup>2</sup>
MPI_KEYVAL_CREATE	MPI-2.0		MPI_COMM_CREATE_KEYVAL
MPI_KEYVAL_FREE	MPI-2.0		MPI_COMM_FREE_KEYVAL
MPI_DUP_FN <sup>3</sup>	MPI-2.0		MPI_COMM_DUP_FN <sup>3</sup>
MPI_NULL_COPY_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_COPY_FN <sup>3</sup>
MPI_NULL_DELETE_FN <sup>3</sup>	MPI-2.0		MPI_COMM_NULL_DELETE_FN <sup>3</sup>
MPI_Copy_function <sup>2</sup>	MPI-2.0		$MPI\_Comm\_copy\_attr\_function^2$
COPY_FUNCTION <sup>2</sup>	MPI-2.0		COMM_COPY_ATTR_FUNCTION <sup>2</sup>
$MPI_Delete_function^2$	MPI-2.0		$MPI\_Comm\_delete\_attr\_function^2$
DELETE_FUNCTION <sup>2</sup>	MPI-2.0		COMM_DELETE_ATTR_FUNCTION <sup>2</sup>
MPI_ATTR_DELETE	MPI-2.0		MPI_COMM_DELETE_ATTR
MPI_ATTR_GET	MPI-2.0		MPI_COMM_GET_ATTR
MPI_ATTR_PUT	MPI-2.0		MPI_COMM_SET_ATTR
MPI_COMBINER_HVECTOR_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_HVECTOR <sup>4</sup>
MPI_COMBINER_HINDEXED_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_HINDEXED <sup>4</sup>
MPI_COMBINER_STRUCT_INTEGER <sup>4</sup>	-	MPI-3.0	MPI_COMBINER_STRUCT <sup>4</sup>
MPI:	MPI-2.2	MPI-3.0	C language binding
MPI_CANCEL for send requests	MPI-4.0		no direct replacement
MPI_INFO_GET	MPI-4.0		MPI_INFO_GET_STRING
MPI_INFO_GET_VALUELEN	MPI-4.0		MPI_INFO_GET_STRING
MPI_T_ERR_INVALID_ITEM	MPI-4.0		MPI_T_ERR_INVALID_INDEX
MPI_SIZEOF	MPI-4.0		storage_size() <sup>5</sup> or c_sizeof()
<sup>1</sup> Predefined datatype.			
<sup>2</sup> Callback prototype definition.			
<sup>3</sup> Predefined callback routine.			
<sup>4</sup> Constant.			
<sup>5</sup> Fortran intrinsic. storage_size() returns the size in bits instead of bytes; see Section 16.3.			
Other entries are regular MPI routines.			

#### Table 2.1: Deprecated and removed constructs

#### 2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an MPI\_ prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare names (identifiers), e.g., for variables, functions, constants, types, or macros, beginning with any prefix of the form MPI\_, where any of the letters are either upper or lower case. To support the profiling interface, programs must not declare functions with names beginning with any prefix of the form PMPI\_, where any of the letters are either upper or lower case.

The definition of named constants, function prototypes, and type definitions must be

 $^{41}$ 

1 supplied in an include file mpi.h.  $\mathbf{2}$ Almost all C functions return an error code. The successful return code will be 3 MPI\_SUCCESS, but failure return codes are implementation dependent. 4 Type declarations are provided for handles to each category of opaque objects. 5Array arguments are indexed from zero. 6 Logical flags are integers with value 0 meaning "false" and a non-zero value meaning  $\overline{7}$ "true." 8 Choice arguments are pointers of type void\*. 9 10 2.6.4 Functions and Macros 11 An implementation is allowed to implement MPI\_WTIME, PMPI\_WTIME, MPI\_WTICK, 12PMPI\_WTICK, MPI\_AINT\_ADD, PMPI\_AINT\_ADD, MPI\_AINT\_DIFF, PMPI\_AINT\_DIFF, 13 and the handle-conversion functions (MPI\_Group\_f2c, etc.) in Section 19.3.4, and no others, 14as macros in C. 1516Advice to implementors. Implementors should document which routines are imple-17 mented as macros. (End of advice to implementors.) 18 19 Advice to users. If these routines are implemented as macros, they will not work 20with the MPI profiling interface. (End of advice to users.) 21222.7 Processes 23 $^{24}$ An MPI program consists of autonomous processes, executing their own code, in an MIMD 25style. The codes executed by each process need not be identical. The processes communicate 26via calls to MPI communication primitives. Typically, each process executes in its own 27address space, although shared-memory implementations of MPI are possible. 28This document specifies the behavior of a parallel program assuming that only MPI 29calls are used. The interaction of an MPI program with other possible means of commu-30 nication, I/O, and process management is not specified. Unless otherwise stated in the  $^{31}$ specification of the standard, MPI places no requirements on the result of its interaction 32 with external mechanisms that provide similar or equivalent functionality. This includes, 33 but is not limited to, interactions with external mechanisms for process control, shared and 34remote memory access, file system access and control, interprocess communication, process 35 signaling, and terminal I/O. High quality implementations should strive to make the results 36 of such interactions intuitive to users, and attempt to document restrictions where deemed 37 necessary. 38

Advice to implementors. Implementations that support such additional mechanisms for functionality supported within MPI are expected to document how these interact with MPI. (*End of advice to implementors.*)

The interaction of MPI and threads is defined in Section 11.6.

### 2.8 Error Handling

39

40

41

42

43 44 45

46

<sup>47</sup> MPI provides the user with reliable message transmission. A message sent is always re-<sup>48</sup> ceived correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with **transmission failures** in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, and to reflect only unrecoverable transmission failures. Whenever possible, such failures will be reflected as errors in the relevant communication call.

Similarly, MPI itself provides no mechanisms for handling MPI **process failures**, that is, when an MPI process unexpectedly and permanently stops communicating (e.g., a software or hardware crash results in an MPI process terminating unexpectedly).

Of course, MPI programs may still be erroneous. A **program error** can occur when an MPI call is made with an incorrect argument (non-existing destination in a send operation, buffer too small in a receive operation, etc.). This type of error would occur in any implementation. In addition, a **resource error** may occur when a program exceeds the amount of available system resources (number of pending messages, system buffers, etc.). The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

19In C and Fortran, almost all MPI calls return a code that indicates successful completion 20of the operation. Whenever possible, MPI calls return an error code if an error occurred 21during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides 22mechanisms for users to change this default and to handle recoverable errors. The user may 23 $^{24}$ specify that no error is fatal, and handle error codes returned by MPI calls by themselves. 25Also, the user may provide user-defined error-handling routines, which will be invoked 26whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 9.3. 27

28Several factors limit the ability of MPI calls to return with meaningful error codes 29when an error occurs. MPI may not be able to detect some errors; other errors may be too 30 expensive to detect in normal execution mode; some faults (e.g., memory faults) may corrupt 31 the state of the MPI library and its outputs; finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller. On the other hand, some 32 33 errors may be detected after the associated operation has completed; some errors may not 34have a communicator, window, or file on which an error may be raised. In such cases, these errors will be raised on the communicator MPI\_COMM\_SELF when using the World 35 Model (see Section 11.2). When MPI\_COMM\_SELF is not initialized (i.e., before MPI\_INIT 36 37 / MPI\_INIT\_THREAD, after MPI\_FINALIZE, or when using the Sessions Model exclusively) the error raises the **initial error handler** (set during the launch operation, see 11.8.4). 38 39 The Sessions Model is described in Section 11.3.

An example of such a case arises because of the nature of asynchronous communications: 40 41 MPI calls may initiate operations that continue asynchronously after the call returned. Thus, 42the operation may return with a code indicating successful completion, yet later cause an error to be raised. If there is a subsequent call that relates to the same operation (e.g., a 4344call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the 4546error may occur after all calls that relate to the operation have completed, so that no error 47value can be used to indicate the nature of the error (e.g., an error on the receiver in a send 48 with the ready mode).

1 2

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8 9

10

11

12

13

14

15

16

17

<sup>1</sup> This document does not specify the state of a computation after an erroneous MPI call <sup>2</sup> has occurred. The desired behavior is that a relevant error code be returned, and the effect <sup>3</sup> of the error be localized to the greatest possible extent. E.g., it is highly desirable that an <sup>4</sup> erroneous receive call will not cause any part of the receiver's memory to be overwritten, <sup>5</sup> beyond the area specified for receiving the message.

<sup>6</sup> Implementations may go beyond this document in supporting in a meaningful manner <sup>7</sup> MPI calls that are defined here to be erroneous. For example, MPI specifies strict type <sup>8</sup> matching rules between matching send and receive operations: it is erroneous to send a <sup>9</sup> floating point variable and receive an integer. Implementations may go beyond these type <sup>10</sup> matching rules, and provide automatic type conversion in such situations. It will be helpful <sup>11</sup> to generate warnings for such nonconforming behavior.

12 13

14

15

MPI defines a way for users to create new error codes as defined in Section 9.5.

2.9 Implementation Issues

There are a number of areas where an MPI implementation may interact with the operating environment and system. While MPI does not mandate that any services (such as signal handling) be provided, it does strongly suggest the behavior to be provided if those services are available. This is an important point in achieving portability across platforms that provide the same set of services.

21 22

23

43

#### 2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment
 (such as write in Fortran and printf and malloc in ISO C) and are executed after
 MPI\_INIT and before MPI\_FINALIZE operate independently and that their *completion* is
 independent of the action of other processes in an MPI program.

Note that this in no way prevents the creation of library routines that provide parallel
 services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI\_COMM\_WORLD (assuming that
 printf is available at the executing nodes).

```
32
33 int commworld_rank;
```

```
MPI_Init((void *)0, (void *)0);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &commworld_rank);
```

```
if (commworld_rank == 0) printf("Starting program\n");
```

MPI\_Finalize();

<sup>38</sup> The corresponding Fortran programs are also expected to complete.

<sup>39</sup> An example of what is *not* required is any particular ordering of the action of these <sup>40</sup> routines when called by several tasks. For example, MPI makes neither requirements nor <sup>41</sup> recommendations for the output from the following program (again assuming that I/O is <sup>42</sup> available at the executing nodes).

```
44 MPI_Comm_rank(MPI_COMM_WORLD, &commworld_rank);
```

printf("Output from MPI process where commworld\_rank=%d\n", commworld\_rank);

In addition, calls that fail because of resource exhaustion or other error are not con sidered a violation of the requirements here (however, they are required to complete, just
 not to complete successfully).

#### 2.9.2 Interaction with Signals

MPI does not specify the interaction of processes with signals and does not require that MPI be signal safe. The implementation may reserve some signals for its own use. It is required that the implementation document which signals it uses, and it is strongly recommended that it not use SIGALRM, SIGFPE, or SIGIO. Implementations may also prohibit the use of MPI calls from within signal handlers.

In multithreaded environments, users can avoid conflicts between signals and the MPI library by catching signals only on threads that do not execute MPI calls. High quality single-threaded implementations will be signal safe: an MPI call suspended by a signal will resume and complete normally after the signal is handled.

### 2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Many of the examples have been compiled by tools that extract the examples from the source files for the MPI standard. However, the examples have not been carefully checked or verified.

 $\overline{7}$ 

## Chapter 3

# **Point-to-Point Communication**

#### 3.1 Introduction

Sending and receiving of *messages* by processes is the basic MPI communication mechanism. The basic point-to-point communication operations are *send* and *receive*. Their use is illustrated in Example 3.1.

```
Example 3.1 A simple 'hello world' example usage of point-to-point communication.
#include "mpi.h"
int main(int argc, char *argv[])
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                      /* code for process zero */
  if (myrank == 0)
  {
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1) /* code for process one */
  ſ
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received :%s:\n", message);
  }
  MPI_Finalize();
  return 0;
}
```

In Example 3.1, process zero (myrank = 0) sends a *message* to process one using the *send* operation MPI\_SEND. The operation specifies a *send buffer* in the sender memory from which the *message data* is taken. In the example above, the send buffer consists of the storage containing the variable **message** in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send

1 operation. The message sent will contain the 13 characters of this variable. In addition,  $\mathbf{2}$ the send operation associates an *envelope* with the message. This *envelope* specifies the 3 message destination and contains distinguishing information that can be used by the *receive* 4 operation to select a particular message. The last three parameters of the send operation,  $\mathbf{5}$ along with the rank of the sender, specify the *envelope* for the message sent. Process one 6 (myrank = 1) receives this message with the *receive* operation MPI\_RECV. The message to 7be received is selected according to the value of its *envelope*, and the *message data* is stored 8 into the receive buffer. In the example above, the receive buffer consists of the storage 9 containing the string **message** in the memory of process one. The first three parameters 10 of the receive operation specify the location, size and type of the receive buffer. The next 11three parameters are used for selecting the incoming message. The last parameter is used 12to return information on the message just received.

<sup>13</sup> The next sections describe the blocking send and receive operations. We discuss send, <sup>14</sup> receive, blocking communication semantics, type matching requirements, type conversion in <sup>15</sup> heterogeneous environments, and more general communication modes. Nonblocking com-<sup>16</sup> munication is addressed next, followed by probing and cancelling a message, channel-like <sup>17</sup> constructs and send-receive operations, ending with a description of the "dummy" process, <sup>18</sup> MPI\_PROC\_NULL.

19 20

21 22

23

24 25 26

27

41

#### 3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the **blocking send** procedure is given below.

MPI\_SEND(buf, count, datatype, dest, tag, comm)

28	IN	buf	initial address of send buffer (choice)	
29 30 31	IN	count	number of elements in send buffer (non-negative integer)	
32	IN	datatype	datatype of each send buffer element (handle)	
33	IN	dest	rank of destination (integer)	
34	IN	tag	message tag (integer)	
35 36	IN	comm	communicator (handle)	
37				
38	B C binding			

int MPI\_Send(const void \*buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)

int MPI\_Send\_c(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)

44 Fortran 2008 binding

```
    MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Send(buf, count, datatype, dest, tag, comm, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: dest, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
                                                                                14
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

The blocking semantics of this call are described in Section 3.4.

#### 3.2.2 Message Data

The send buffer specified by the MPI\_SEND procedure consists of count successive entries of the type indicated by datatype, starting with the entry at address buf. Note that we specify the message length in terms of number of *elements*, not number of *bytes*. The former is machine independent and closer to the application level.

The data part of the message consists of a sequence of **count** values, each of the type indicated by datatype. count may be zero, in which case the data part of the message is empty. The **basic datatypes** that can be specified for message data values correspond to the basic datatypes of the host language. Possible values of this argument for Fortran and the corresponding Fortran types are listed in Table 3.1. Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

#### Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

The datatypes MPI\_BYTE and MPI\_PACKED do not correspond to a Fortran or C datatype. A value of type MPI\_BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type MPI\_PACKED is explained in Section 5.2.

1

 $\mathbf{2}$ 

4

5

6

7

9

10 11

1213

15

1617

18 19

20

21

22

23

24

25

26

27

28

41 42

43

44

45

46

47

1	MPI datatype	C datatype
2	MPI_CHAR	char
3		(treated as printable character)
4	MPI_SHORT	signed short int
5	MPI_INT	signed int
6	MPI_LONG	signed long int
7	MPI_LONG_LONG_INT	signed long long int
8	MPI_LONG_LONG (as a synonym)	signed long long int
9	MPI_SIGNED_CHAR	signed char
10		(treated as integral value)
11	MPI_UNSIGNED_CHAR	unsigned char
12		(treated as integral value)
13	MPI_UNSIGNED_SHORT	unsigned short int
14	MPI_UNSIGNED	unsigned int
15	MPI_UNSIGNED_LONG	unsigned long int
16	MPI_UNSIGNED_LONG_LONG	unsigned long long int
17	MPI_FLOAT	float
18	MPI_DOUBLE	double
19	MPI_LONG_DOUBLE	long double
20	MPI_WCHAR	wchar_t
21		(defined in <stddef.h>)</stddef.h>
22		(treated as printable character)
23	MPI_C_BOOL	_Bool
24	MPI_INT8_T	int8_t
25	MPI_INT16_T	int16_t
26	MPI_INT32_T	int32_t
27	MPI_INT64_T	int64_t
28	MPI_UINT8_T	uint8_t
29	MPI_UINT16_T	uint16_t
30	MPI_UINT32_T	uint32_t
31	MPI_UINT64_T	uint64_t
32	MPI_C_COMPLEX	float _Complex
33	MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
34	MPI_C_DOUBLE_COMPLEX	double _Complex
35	MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
36	MPI_BYTE	
37	MPI_PACKED	
38		
39		

- 40
- 41

Table 3.2: Predefined MPI datatypes corresponding to C datatypes

MPI requires support of these datatypes, which match the basic datatypes of Fortran
 and ISO C. Additional MPI datatypes should be provided if the host language has additional
 datatypes<sup>1</sup>: MPI\_DOUBLE\_COMPLEX for double precision complex in Fortran declared to
 be of type DOUBLE COMPLEX; MPI\_REAL2, MPI\_REAL4, MPI\_REAL8, and

<sup>46</sup> <sup>1</sup>These types, such as DOUBLE COMPLEX and INTEGER\*4, are not specified by any Fortran standard but are extensions commonly accepted by Fortran compilers.
 <sup>48</sup>

MPI datatype	C datatype	Fortran datatype
MPI_AINT	MPI_Aint	INTEGER(KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	MPI_Offset	INTEGER(KIND=MPI_OFFSET_KIND)
MPI_COUNT	MPI_Count	INTEGER(KIND=MPI_COUNT_KIND)

Table 3.3: Predefined MPI datatypes corresponding to both C and Fortran datatypes

MPI\_REAL16 for Fortran reals, declared to be of type REAL\*2, REAL\*4, REAL\*8, and REAL\*16, respectively; MPI\_INTEGER1, MPI\_INTEGER2, MPI\_INTEGER4, and MPI\_INTEGER8 for Fortran integers, declared to be of type INTEGER\*1, INTEGER\*2, INTEGER\*4, and INTEGER\*8, respectively; MPI\_COMPLEX4, MPI\_COMPLEX8, MPI\_COMPLEX16, and MPI\_COMPLEX32 for complex numbers in Fortran declared to be of type COMPLEX\*4, COMPLEX\*8, COMPLEX\*16, and COMPLEX\*32, respectively; etc.

*Rationale.* One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (*End of rationale.*)

The datatypes MPI\_AINT, MPI\_OFFSET, and MPI\_COUNT correspond to the MPI-defined C types MPI\_Aint, MPI\_Offset, and MPI\_Count and their Fortran equivalents INTEGER(KIND= MPI\_ADDRESS\_KIND), INTEGER(KIND=MPI\_OFFSET\_KIND), and INTEGER(KIND= MPI\_COUNT\_KIND). This is described in Table 3.3. All predefined datatype handles are available in all language bindings. See Sections 19.3.6 and 19.3.10 on page 846 and 854 for information on interlanguage communication with these types.

If there is an accompanying C++ compiler then the datatypes in Table 3.4 are also supported in C and Fortran.

MPI datatype	C++ datatype
MPI_CXX_BOOL	bool
MPI_CXX_FLOAT_COMPLEX	<pre>std::complex<float></float></pre>
MPI_CXX_DOUBLE_COMPLEX	<pre>std::complex<double></double></pre>
MPI_CXX_LONG_DOUBLE_COMPLEX	<pre>std::complex<long double=""></long></pre>

Table 3.4: Predefined MPI datatypes corresponding to C++ datatypes

#### 3.2.3 Message Envelope

In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the **message envelope**. These fields are

1	communicator
2	
3	The message source is implicitly determined by the identity of the message sender. The
4	other fields are specified by arguments in the send procedure.
5	The message destination is specified by the dest argument.
6	The integer-valued message tag is specified by the tag argument. This integer can be
7	used by the program to distinguish different types of messages. The range of valid tag
8	values is $0, \ldots, UB$ , where the value of UB is implementation dependent. It can be found by
9 10	querying the value of the attribute MPI_TAG_UB, as described in Chapter 9. MPI requires that UB be no less than 32767.
11	The comm argument specifies the <i>communicator</i> that is used for the send operation.
12	Communicators are explained in Chapter 7; below is a brief summary of their usage.
13	A communicator specifies the communication context for a communication operation.
14	Each communication context provides a separate "communication universe": messages are
15	always received within the context they were sent, and messages sent in different contexts
16	do not interfere.
17	The communicator also specifies the set of processes that share this communication
18	context. This <i>process group</i> is ordered and processes are identified by their rank within this
19	group. Thus, the range of valid values for dest is $0, \ldots, n-1 \cup \{MPI\_PROC\_NULL\}$ , where
20	n is the number of processes in the group. (If the communicator is an inter-communicator,
21	then destinations are identified by their rank in the remote group. See Chapter 7.)
22	When using the World Model (see Section 11.2), a predefined communicator
23	MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that
24	are accessible after MPI initialization and processes are identified by their rank in the group
25	of MPI_COMM_WORLD.
26	Advice to users. Users that are comfortable with the notion of a flat name space
27	for processes, and a single communication context, as offered by most existing com-
28	munication libraries, need only use the World Model for MPI initialization, and the
29	predefined variable MPI_COMM_WORLD as the comm argument. This will allow com-
$30 \\ 31$	munication with all the processes available at initialization time.
32	Users may define new communicators, as explained in Chapter 7. Communicators
33	provide an important encapsulation mechanism for libraries and modules. They allow
34	modules to have their own disjoint communication universe and their own process
35	numbering scheme. (End of advice to users.)
36	numbering benefite. (Drive of warree to works.)
37	Advice to implementors. The message envelope would normally be encoded by a
38	fixed-length message header. However, the actual encoding is implementation depen-
39	dent. Some of the information (e.g., source or destination) may be implicit, and need
40	not be explicitly carried by messages. Also, processes may be identified by relative
41	ranks, or absolute ids, etc. (End of advice to implementors.)
42	
43	3.2.4 Blocking Receive
44	The syntax of the blocking receive presedure is given below
45	The syntax of the <b>blocking receive</b> procedure is given below.
46	
47	
48	

MPI_REC	CV(buf, count, datatyp	pe, source, tag, comm, status)	1
OUT	buf	initial address of receive buffer (choice)	2 3
IN	count	number of elements in receive buffer (non-negative integer)	4 5
IN	datatype	datatype of each receive buffer element (handle)	6
IN	source	rank of source or $MPI_ANY_SOURCE$ (integer)	7
IN	tag	message tag or MPI_ANY_TAG (integer)	8 9
IN	comm	communicator (handle)	10
OUT	status	status object (status)	11 12
	Recv(void *buf, i int tag, MP]	nt count, MPI_Datatype datatype, int source, [_Comm comm, MPI_Status *status) MPI_Count count, MPI_Datatype datatype,	13 14 15 16 17
_	-	int tag, MPI_Comm comm, MPI_Status *status)	18
MPI_Recv TYPE INTE TYPE TYPE TYPE	E(*), DIMENSION( EGER, INTENT(IN) : E(MPI_Datatype), I E(MPI_Comm), INTEN E(MPI_Status) :: s	: count, source, tag NTENT(IN) :: datatype T(IN) :: comm	20 21 22 23 24 25 26 27
TYPI INTI TYPI INTI TYPI TYPI	E(*), DIMENSION( EGER(KIND=MPI_COUN E(MPI_Datatype), I EGER, INTENT(IN) : E(MPI_Comm), INTEN E(MPI_Status) :: s	T_KIND), INTENT(IN) :: count NTENT(IN) :: datatype : source, tag T(IN) :: comm	28 29 30 31 32 33 34 35
MPI_RECV <typ INTE</typ 	De> BUF(*) EGER COUNT, DATATY IERROR blocking semantics o	TYPE, SOURCE, TAG, COMM, STATUS, IERROR) PE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), f this call are described in Section 3.4.	36 37 38 39 40 41 42 43
Fortran MPI_RECV <typ INTE</typ 	binding /(BUF, COUNT, DATA be> BUF(*) EGER COUNT, DATATY IERROR blocking semantics o	TYPE, SOURCE, TAG, COMM, STATUS, IERROR) PE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),	

g ıg type specified by datatype, starting at address buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

If a message that is shorter than the receive buffer arrives, then only those locations corresponding to the (shorter) message are modified.

Advice to users. The MPI\_PROBE function described in Section 3.8 can be used to receive messages of unknown length. (*End of advice to users.*)

Advice to implementors. Even though no specific behavior is mandated by MPI for erroneous programs, the recommended handling of overflow situations is to return in status information about the source and tag of the incoming message. The receive procedure will return an error code. A quality implementation will also ensure that no memory that is outside the receive buffer will ever be overwritten.

In the case of a message shorter than the receive buffer, MPI is quite strict in that it allows no modification of the other locations. A more lenient statement would allow for some optimizations but this is not allowed. The implementation must be ready to end a copy into the receiver memory exactly at the end of the receive buffer, even if it is an odd address. (*End of advice to implementors.*)

14The selection of a message by a receive operation is governed by the value of the 15message envelope. A message can be received by a receive operation if its envelope matches 16the source, tag and comm values specified by the receive operation. The receiver may specify 17a wildcard MPI\_ANY\_SOURCE value for source, and/or a wildcard MPI\_ANY\_TAG value for 18 tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value 19for comm. Thus, a message can be received by a receive operation only if it is addressed 20to the receiving process, has a matching communicator, has matching source unless source 21= MPI\_ANY\_SOURCE in the pattern, and has a matching tag unless tag = MPI\_ANY\_TAG in 22 the pattern. 23

The message tag is specified by the tag argument of the receive operation. The argument source, if different from MPI\_ANY\_SOURCE, is specified as a rank within the process group associated with that same communicator (remote process group, for intercommunicators). Thus, the range of valid values for the source argument is  $\{0, ..., n-1\} \cup$ {MPI\_ANY\_SOURCE} $\cup$ {MPI\_PROC\_NULL}, where *n* is the number of processes in this group.

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify a unique receiver. This matches a "push" communication mechanism, where data transfer is effected by the sender (rather than a "pull" mechanism, where data transfer is effected by the receiver).

Source = destination is allowed, that is, a process can send a message to itself. However, it is unsafe to do so with the blocking send and receive operations described above, since this may lead to deadlock. See Section 3.5.

Advice to implementors. Message context and other communicator information can be implemented as an additional tag field. It differs from the regular message tag in that wild card matching is not allowed on this field, and that value setting for this field is controlled by communicator manipulation functions. (*End of advice to implementors.*)

The use of dest = MPI\_PROC\_NULL or source = MPI\_PROC\_NULL to define a "dummy" destination or source in any send or receive call is described in Section 3.10.

3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function

1

2

3

4

5

6

7

8

9

10

11

12

13

34

35

36 37

38

39

40

 $41 \\ 42$ 

43

44 45

(see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI\_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR; the structure may contain additional fields. Thus,

status.MPI\_SOURCE, status.MPI\_TAG, and status.MPI\_ERROR contain the source, tag, and error code, respectively, of the received message.

In Fortran with USE mpi or INCLUDE 'mpif.h', status is an array of INTEGERS of size MPI\_STATUS\_SIZE. The constants MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR are the indices of the entries that store the source, tag, and error fields. Thus, status(MPI\_SOURCE), status(MPI\_TAG), and status(MPI\_ERROR) contain, respectively, the source, tag, and error code of the received message.

With Fortran USE mpi\_f08, status is defined as the Fortran BIND(C) derived type TYPE(MPI\_Status) containing three public INTEGER fields named MPI\_SOURCE, MPI\_TAG, and MPI\_ERROR. TYPE(MPI\_Status) may contain additional, implementation-specific fields. Thus, status%MPI\_SOURCE, status%MPI\_TAG, and status%MPI\_ERROR contain the source, tag, and error code of a received message respectively. Additionally, within both the mpi and the mpi\_f08 modules, the constants MPI\_STATUS\_SIZE, MPI\_SOURCE, MPI\_TAG, MPI\_TAG, MPI\_ERROR, and TYPE(MPI\_Status) are defined to allow conversion between both status representations. Conversion routines are provided in Section 19.3.5.

*Rationale.* The Fortran TYPE(MPI\_Status) is defined as a BIND(C) derived type so that it can be used at any location where the status integer array representation can be used, e.g., in user defined common blocks. (*End of rationale.*)

*Rationale.* It is allowed to have the same name (e.g., MPI\_SOURCE) defined as a constant (e.g., Fortran parameter) and as a field of a derived type. (*End of rationale.*)

In general, message-passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 3.7.5 which return multiple statuses. The field is updated if and only if such function returns with an error code of MPI\_ERR\_IN\_STATUS.

*Rationale.* The error field in status is not needed for calls that return only one status, such as MPI\_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure. (*End of rationale.*)

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to MPI\_GET\_COUNT is required to "decode" this information.  $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

1	MPI_GET	_COUNT(status, datatype, cour	nt)
2	IN	status	return status of receive operation (status)
$\frac{3}{4}$	IN	datatype	datatype of each receive buffer entry (handle)
5	OUT	count	number of received entries (integer)
6			
7	C bindin	g	
8 9	int MPI_(		s *status, MPI_Datatype datatype,
10		int *count)	
11 12	int MPI_(	Get_count_c(const MPI_Stat MPI_Count *count)	cus *status, MPI_Datatype datatype,
13	Fortran 2	2008 binding	
14 15		count(status, datatype, co	
16		(MPI_Status), INTENT(IN) :	
17		(MPI_Datatype), INTENT(IN) GER, INTENT(OUT) :: count	:: datatype
18		GER, OPTIONAL, INTENT(OUT)	:: ierror
19 20	MPT Cet (	count(status, datatype, co	unt ierror) I(c)
20		(MPI_Status), INTENT(IN) :	
22	TYPE	(MPI_Datatype), INTENT(IN)	:: datatype
23		GER(KIND=MPI_COUNT_KIND),	
24 25	INTE	GER, OPTIONAL, INTENT(OUT)	:: ierror
25 26	Fortran l	0	
27		COUNT(STATUS, DATATYPE, CO	
28			E), DATATYPE, COUNT, IERROR
29			ed. (Again, we count <i>entries</i> , each of type datatype,
30 31			d match the argument provided by the receive call er of entries received exceeds the limits of the count
32			the value of count to MPI_UNDEFINED. There are
33	other situa	ations where the value of <b>count</b>	can be set to MPI_UNDEFINED; see Section $5.1.11$ .
34	Data	ionala Somo mozzaro pazzin	bipropries use MOUT count tag and course angu
35 36			g libraries use INOUT count, tag and source argu- becify the selection criteria for incoming messages
37			ues of the received message. The use of a separate
38		<u> </u>	at are often attached with INOUT argument (e.g.,
39		0	as the tag in a receive). Some libraries use calls
40		× •	essage received." This is not thread safe.
41 42			• MPI_GET_COUNT so as to improve performance.
43		0 0	out counting the number of elements it contains, needed. Also, this allows the same function to be
44			or MPI_IPROBE. With a status from MPI_PROBE
45	or N	<b>IPI_IPROBE</b> , the same datatyp	hes are allowed as in a call to $MPI_RECV$ to receive
46 47	this	message. (End of rationale.)	
48			

The value returned as the count argument of MPI\_GET\_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI\_UNDEFINED is returned.

*Rationale.* Zero-length datatypes may be created in a number of cases. An important case is MPI\_TYPE\_CREATE\_DARRAY, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use MPI\_GET\_COUNT to check the status. (*End of rationale.*)

Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with MPI\_GET\_COUNT and the receive. (*End of advice to users.*)

All send and receive operations use the buf, count, datatype, source, dest, tag, comm, and status arguments in the same way as the blocking MPI\_SEND and MPI\_RECV procedures described in this section.

#### 3.2.6 Passing MPI\_STATUS\_IGNORE for Status

Every call to MPI\_RECV includes a status argument, wherein the system can return details about the message received. There are also a number of other MPI calls where status is returned. An object of type MPI\_Status is not an MPI opaque object; its structure is declared in mpi.h and mpif.h, and it exists in the user's program. In many cases, application programs are constructed so that it is unnecessary for them to examine the status fields. In these cases, it is a waste for the user to allocate a status object, and it is particularly wasteful for the MPI implementation to fill in fields in this object.

To cope with this problem, there are two predefined constants, MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE, which when passed to a receive, probe, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that MPI\_STATUS\_IGNORE is not a special type of MPI\_Status object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI\_Status.

MPI\_STATUS\_IGNORE, and the array version MPI\_STATUSES\_IGNORE, can be used everywhere a status argument is passed to a receive, wait, or test function. MPI\_STATUS\_IGNORE cannot be used when status is an IN argument. Note that in Fortran MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE are objects like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

In general, this optimization can apply to all functions for which status or an array of statuses is an OUT argument. Note that this converts status into an INOUT argument. The functions that can be passed MPI\_STATUS\_IGNORE are all the various forms of MPI\_RECV, MPI\_PROBE, MPI\_TEST, and MPI\_WAIT, as well as MPI\_REQUEST\_GET\_STATUS. When an array is passed, as in the MPI\_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI\_STATUSES\_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI\_ERR\_IN\_STATUS even when MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE has been passed to that function.

MPI\_STATUS\_IGNORE and MPI\_STATUSES\_IGNORE are not required to have the same values in C and Fortran.

 $\overline{7}$ 

It is not allowed to have some of the statuses in an array of statuses for MPI\_{TEST|WAIT}{ALL|SOME} functions set to MPI\_STATUS\_IGNORE; one either specifies ignoring *all* of the statuses in such a call with MPI\_STATUSES\_IGNORE, or *none* of them by passing normal statuses in all positions in the array of statuses.

5 6 7

## 3.2.7 Blocking Send-Receive

The send-receive operations combine in one operation the sending of a message to one 8 destination and the receiving of another message, from another process. The two (source 9 and destination) are possibly the same. A send-receive operation is very useful for executing 10 a shift operation across a chain of processes. If blocking sends and receives are used for such 11 a shift, then one needs to order the sends and receives correctly (for example, even processes 12send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies 13 that may lead to *deadlock*. When a send-receive operation is used, the communication 14 subsystem takes care of these issues. The send-receive operation can be used in conjunction 15with the procedures described in Chapter 8 in order to perform shifts on various logical 16topologies. Also, a send-receive operation is useful for implementing remote procedure 17calls. 18

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

22 23

24

MPI_SENDRECV(sendbuf, sendcount, sendtyp	e, dest, sendtag, recvbuf, recvcount, recvtype,
source, recvtag, comm, status	)

25			- ,
26	IN	sendbuf	initial address of send buffer (choice)
27 28	IN	sendcount	number of elements in send buffer (non-negative integer)
29 30	IN	sendtype	type of elements in send buffer (handle)
31	IN	dest	rank of destination (integer)
32	IN	sendtag	send tag (integer)
33 34	OUT	recvbuf	initial address of receive buffer (choice)
35 36	IN	recvcount	number of elements in receive buffer (non-negative integer)
37	IN	recvtype	type of elements receive buffer element (handle)
38 39	IN	source	rank of source or $MPI_ANY_SOURCE$ (integer)
40	IN	recvtag	receive tag or $MPI_ANY_TAG$ (integer)
41	IN	comm	communicator (handle)
42 43	OUT	status	status object (status)
44			
45	C bindi		

int MPI\_Sendrecv(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, int dest, int sendtag, void \*recvbuf, int recvcount,

by a join of these two threads.

1 MPI\_Datatype recvtype, int source, int recvtag, MPI\_Comm comm, 2 MPI\_Status \*status) 3 int MPI\_Sendrecv\_c(const void \*sendbuf, MPI\_Count sendcount, 4 MPI\_Datatype sendtype, int dest, int sendtag, void \*recvbuf, 5MPI\_Count recvcount, MPI\_Datatype recvtype, int source, 6 int recvtag, MPI\_Comm comm, MPI\_Status \*status) 7 8 Fortran 2008 binding 9 MPI\_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, 10 recvcount, recvtype, source, recvtag, comm, status, ierror) 11 TYPE(\*), DIMENSION(..), INTENT(IN) :: sendbuf INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source, 1213 recvtag 14TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 15TYPE(\*), DIMENSION(..) :: recvbuf 16TYPE(MPI\_Comm), INTENT(IN) :: comm 17TYPE(MPI\_Status) :: status 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 MPI\_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, 20recvcount, recvtype, source, recvtag, comm, status, ierror) 21!( c) 22 TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf 23INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 24TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 25INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag 26TYPE(\*), DIMENSION(..) :: recvbuf 27TYPE(MPI\_Comm), INTENT(IN) :: comm 28TYPE(MPI\_Status) :: status 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 31Fortran binding 32 MPI\_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 33 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR) 34 <type> SENDBUF(\*), RECVBUF(\*) 35 INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 36 SOURCE, RECVTAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR 37 Execute a blocking send-receive operation. Both send and receive use the same com-38 municator, but possibly different tags. The send buffer and receive buffers must be disjoint, 39 and may have different lengths and datatypes. 40 The semantics of a send-receive operation is what would be obtained if the caller forked 41 two concurrent threads, one to execute the send, and one to execute the receive, followed 42

```
1
     MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm,
\mathbf{2}
                    status)
3
       INOUT
                buf
                                            initial address of send and receive buffer (choice)
4
       IN
                                            number of elements in send and receive buffer
                count
5
                                            (non-negative integer)
6
7
       IN
                datatype
                                            type of elements in send and receive buffer (handle)
8
       IN
                dest
                                            rank of destination (integer)
9
                sendtag
                                            send message tag (integer)
       IN
10
11
       IN
                source
                                            rank of source or MPI_ANY_SOURCE (integer)
12
       IN
                 recvtag
                                            receive message tag or MPI_ANY_TAG (integer)
13
       IN
                comm
                                            communicator (handle)
14
15
       OUT
                                            status object (status)
                status
16
17
     C binding
18
     int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype,
19
                    int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
20
                    MPI_Status *status)
21
     int MPI_Sendrecv_replace_c(void *buf, MPI_Count count,
22
                    MPI_Datatype datatype, int dest, int sendtag, int source,
23
                    int recvtag, MPI_Comm comm, MPI_Status *status)
^{24}
25
     Fortran 2008 binding
26
     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
27
                    comm, status, ierror)
28
         TYPE(*), DIMENSION(..) :: buf
29
         INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         TYPE(MPI_Status) :: status
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
35
                    comm, status, ierror) !(_c)
36
         TYPE(*), DIMENSION(..) :: buf
37
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         TYPE(MPI_Status) :: status
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     Fortran binding
45
     MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
46
                    COMM, STATUS, IERROR)
47
          <type> BUF(*)
48
```

INTEGER	COUNT,	DATATYPE,	DEST,	SENDTAG,	SOURCE,	RECVTAG,	COMM,
	STATU	JS(MPI_STAT	rus_siz	ZE), IERRO	DR		

Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.

Advice to implementors. Additional intermediate buffering is needed for the "replace" variant. (End of advice to implementors.)

# 3.3 Datatype Matching and Data Conversion

#### 3.3.1 Type Matching Rules

One can think of message transfer as consisting of the following three phases.

- 1. Data is pulled out of the send buffer and a message is assembled.
- 2. A message is transferred from sender to receiver.
- 3. Data is pulled from the incoming message and disassembled into the receive buffer.

**Type matching** has to be observed at each of these three phases: The type of each variable in the sender buffer has to match the type specified for that entry by the send operation; the type specified by the send operation has to match the type specified by the receive operation; and the type of each variable in the receive buffer has to match the type specified for that entry by the receive operation. A program that fails to observe these three rules is *erroneous*.

To define type matching more precisely, we need to deal with two issues: matching of types of the host language with types specified in communication operations; and matching of types at sender and receiver.

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI\_INTEGER matches MPI\_INTEGER, MPI\_REAL matches MPI\_REAL, and so on. There is one exception to this rule, discussed in Section 5.2: the type MPI\_PACKED can match any other type.

The type of a variable in a host program matches the type specified in the commu-nication operation if the datatype name used by that operation corresponds to the basic type of the host program variable. For example, an entry with type name MPI\_INTEGER matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with type name MPI\_BYTE or MPI\_PACKED can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type MPI\_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 5.2. The type MPI\_BYTE allows one to transfer the binary value of a byte in memory unchanged. 

To summarize, the type matching rules fall into the three categories below.

• Communication of typed values (e.g., with datatype different from MPI\_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.

- Communication of untyped values (e.g., of datatype MPI\_BYTE), where both sender and receiver use the datatype MPI\_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where MPI\_PACKED is used.

The following examples illustrate the first two cases.

```
Example 3.2 Sender and receiver specify matching types.
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

This code is correct if both **a** and **b** are real arrays of size  $\geq 10$ . (In Fortran, it might be correct to use this code even if **a** or **b** have size < 10: e.g., when **a(1)** can be equivalenced to an array with ten reals.)

**Example 3.3** Sender and receiver do not specify matching types.

```
! ----- THIS EXAMPLE IS ERRONEOUS -----
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is *erroneous*, since sender and receiver do not provide matching datatype arguments.

**Example 3.4** Sender and receiver specify communication of untyped values.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

This code is correct, irrespective of the type and size of **a** and **b** (unless this results in an out of bounds memory access).

Advice to users. If a buffer of type MPI\_BYTE is passed as an argument to MPI\_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

16 17

18

19

20 21

22 23

24

25

26

27

28

29

30 31

32

33 34

35 36

37

38

39

40

41

42 43

 $^{44}$ 

 $45 \\ 46$ 

47

layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI\_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)

Type MPI\_CHARACTER

The type MPI\_CHARACTER matches one character of a Fortran variable of type CHARACTER, rather than the entire character string stored in the variable. Fortran variables of type CHARACTER or substrings are transferred as if they were arrays of characters. This is illustrated in the example below.

Example 3.5 Transfer of Fortran CHARACTERs.

```
CHARACTER*10 a
CHARACTER*10 b
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(a, 5, MPI_CHARACTER, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(b(6:10), 5, MPI_CHARACTER, 0, tag, comm, status, ierr)
END IF
```

The last five characters of string **b** at process 1 are replaced by the first five characters of string **a** at process 0.

*Rationale.* The alternative choice would be for MPI\_CHARACTER to match a character of arbitrary length. This runs into problems.

A Fortran character variable is a constant length string, with no special termination symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 5.1). If this communicator buffer contains variables of type CHARACTER then the information on their length will not be passed to the MPI routine.

This problem forces us to provide explicit information on character length with the MPI call. One could add a length parameter to the type MPI\_CHARACTER, but this does not add much convenience and the same functionality can be achieved by defining a suitable derived datatype. (*End of rationale.*)

Advice to implementors. Some compilers pass Fortran CHARACTER arguments as a structure with a length and a pointer to the actual string. In such an environment, the MPI call needs to dereference the pointer in order to reach the string. (End of advice to implementors.)

 $^{24}$ 

## 3.3.2 Data Conversion

One of the goals of MPI is to support parallel computations across heterogeneous environments. Communication in a heterogeneous environment may require data conversions. We use the following terminology.

type conversion changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.

**representation conversion** changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical and character values, and to convert a floating point value to the nearest value that can be represented on the target system.

Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

If a value sent in a message is untyped (i.e., of type MPI\_BYTE), then the binary representation of the byte stored at the receiver is identical to the binary representation of the byte loaded at the sender. This holds true, whether sender and receiver run in the same or in distinct environments. No representation conversion is required. (Note that representation conversion may occur when values of type MPI\_CHARACTER or MPI\_CHAR are transferred, for example, from an EBCDIC encoding to an ASCII encoding.)

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment.

Consider the three examples, 3.2–3.4. The first program is correct, assuming that **a** and 30 b are REAL arrays of size > 10. If the sender and receiver execute in different environments,  $^{31}$ then the ten real values that are fetched from the send buffer will be converted to the 32 representation for reals on the receiver site before they are stored in the receive buffer. 33 While the number of real elements fetched from the send buffer equal the number of real 34elements stored in the receive buffer, the number of bytes stored need not equal the number 35 of bytes loaded. For example, the sender may use a four byte representation and the receiver 36 an eight byte representation for reals. 37

38

48

The second program is *erroneous*, and its behavior is undefined.

The third program is correct. The exact same sequence of forty bytes that were loaded from the send buffer will be stored in the receive buffer, even if sender and receiver run in a different environment. The message sent has exactly the same length (in bytes) and the same binary representation as the message received. If **a** and **b** are of different types, or if they are of the same type but different data representations are used, then the bits stored in the receive buffer may encode values that are different from the values they encoded in the send buffer.

<sup>46</sup> Data representation conversion also applies to the *envelope* of a message: source, des-<sup>47</sup> tination and tag are all integers that may need to be converted.

48

1

2

3

4

5 6

7

8

Advice to implementors. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.

Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (*End of advice to implementors.*)

MPI requires support for inter-language communication, e.g., if messages are sent using an MPI procedure from the MPI C language interface and received using an MPI procedure from one of the MPI Fortran language interfaces. The behavior is defined in Section 19.3.

## 3.4 Communication Modes

The send call described in Section 3.2.1 is *blocking*: it does not return until the *message data* and *envelope* have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, message buffering can be expensive, as it entails additional memory-to-memory copying, and it requires the allocation of memory for buffering. MPI offers the choice of several **communication modes** that allow one to control the choice of the communication protocol.

The send call described in Section 3.2.1 uses the **standard** communication mode. In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Thus, a *standard mode send* can be *started* whether or not a matching receive has been posted. It may *complete* before a matching receive is posted. The standard mode send is *non-local*: successful completion of the send operation may depend on the occurrence of a matching receive.

*Rationale.* The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs. Since any system will run out of buffer resources as message sizes are increased, and some implementations may want to provide little buffering, MPI takes the position that correct (and therefore, portable) programs do not rely on system buffering in standard mode. Buffering may improve the performance of a correct program, but it doesn't affect the result of the program. If the user wishes to guarantee a certain amount of buffering, the userprovided buffer system of Section 3.6 should be used, along with the buffered-mode send. (*End of rationale.*)

There are three additional communication modes.

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

1 A **buffered** mode send operation can be started whether or not a matching receive  $\mathbf{2}$ has been posted. It may complete before a matching receive is posted. However, unlike the 3 standard send, this operation is *local*, and its completion does not depend on the occurrence 4 of a matching receive. Thus, if a send is executed and no matching receive is posted, then  $\mathbf{5}$ MPI must buffer the outgoing message, so as to allow the send call to complete. An error will 6 occur if there is insufficient buffer space. The amount of available buffer space is controlled  $\overline{7}$ by the user—see Section 3.6. Buffer allocation by the user may be required for the buffered 8 mode to be effective.

9 A send that uses the **synchronous** mode can be started whether or not a matching 10 receive was posted. However, the send will complete successfully only if a matching receive is 11posted, and the receive operation has started to receive the message sent by the synchronous 12send. Thus, the completion of a synchronous send not only indicates that the send buffer 13can be reused, but it also indicates that the receiver has reached a certain point in its 14execution, namely that it has started executing the matching receive. If both sends and 15receives are blocking operations then the use of the synchronous mode provides synchronous 16communication semantics: a communication does not complete at either end before both 17processes rendezvous at the communication. A send executed in this mode is non-local.

18 A send that uses the **ready** communication mode may be started *only* if the matching 19receive is already posted. Otherwise, the operation is erroneous and its outcome is unde-20fined. On some systems, this allows the removal of a hand-shake protocol that is otherwise 21required and results in improved performance. The completion of the send operation does 22not depend on the status of a matching receive, and merely indicates that the send buffer 23can be reused. A send operation that uses the ready mode has the same semantics as a  $^{24}$ standard send operation, or a synchronous send operation; it is merely that the sender 25provides additional information to the system (namely that a matching receive is already 26posted), that can save some overhead. In a correct program, therefore, a ready send could 27be replaced by a standard send with no effect on the behavior of the program other than 28performance.

Three additional send functions are provided for the three additional communication
 modes. The communication mode is indicated by a one letter prefix: B for buffered, S for
 synchronous, and R for ready.

32 33

34

48

MPI\_BSEND(buf, count, datatype, dest, tag, comm)

35	IN	buf	initial address of send buffer (choice)
36 37 38	IN	count	number of elements in send buffer (non-negative integer)
39	IN	datatype	datatype of each send buffer element (handle)
40	IN	dest	rank of destination (integer)
41 42	IN	tag	message tag (integer)
43	IN	comm	communicator (handle)
44			

#### 45 C binding

<sup>46</sup> int MPI\_Bsend(const void \*buf, int count, MPI\_Datatype datatype, int dest, <sup>47</sup> int tag, MPI\_Comm comm)

int M	PI Bsend c(const void	d *buf, MPI_Count count, MPI_Datatype datatype,	1			
1110 11		t tag, MPI_Comm comm)	2			
Fontn	an 2008 binding		3			
	6	atype, dest, tag, comm, ierror)	4			
	YPE(*), DIMENSION()		5 6			
	NTEGER, INTENT(IN) :	-	7			
	• 1	NTENT(IN) :: datatype	8			
	YPE(MPI_Comm), INTEN		9			
I	INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
MPI_B	send(buf, count, data	atype, dest, tag, comm, ierror) !(_c)	11			
	YPE(*), DIMENSION()		12			
		T_KIND), INTENT(IN) :: count	13 14			
	YPE(MP1_Datatype), 11 NTEGER, INTENT(IN) :	NTENT(IN) :: datatype	14			
	YPE(MPI_Comm), INTEN'		16			
	NTEGER, OPTIONAL, IN		17			
			18			
	an binding	ATYPE, DEST, TAG, COMM, IERROR)	19			
	type> BUF(*)	ATTE, DEST, TAG, COMP, TERROR/	20			
	• 1	PE, DEST, TAG, COMM, IERROR	21 22			
S	end in buffered mode.		23			
		ons in Section 2.4.2, MPI_BSEND is a completing procedure	24			
and the user can re-use all resources given as arguments, including the <i>message data buffer</i> .						
It is also a local procedure because it returns immediately without depending on the exe-						
cution	cution of any MPI procedure in any other MPI process.					
Advice to users. This is one of the exceptions in which a completing and therefore						
		ed procedure is local. ( <i>End of advice to users.</i> )	29 30			
			31			
			32			
MPI_S	SEND(buf, count, dataty	rpe, dest, tag, comm)	33			
IN	buf	initial address of send buffer (choice)	34			
IN	count	number of elements in send buffer (non-negative	35 36			
	count	integer)	37			
IN	datatype	datatype of each send buffer element (handle)	38			
IN	dest	rank of destination (integer)	39			
			40			
IN	tag	message tag (integer)	41			
IN	comm	communicator (handle)	42 43			
<b>a</b> 1 '	1		44			
	C binding					
тпс I,II	<pre>int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>					
	47					
			48			

```
1
     int MPI_Ssend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
\mathbf{2}
                    int dest, int tag, MPI_Comm comm)
3
     Fortran 2008 binding
4
     MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
6
         INTEGER, INTENT(IN) :: count, dest, tag
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) !(_c)
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
13
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         INTEGER, INTENT(IN) :: dest, tag
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     Fortran binding
19
     MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
20
          <type> BUF(*)
21
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
22
23
         Send in synchronous mode.
^{24}
25
     MPI_RSEND(buf, count, datatype, dest, tag, comm)
26
27
       IN
                buf
                                           initial address of send buffer (choice)
28
       IN
                                           number of elements in send buffer (non-negative
                count
29
                                           integer)
30
       IN
                datatype
                                           datatype of each send buffer element (handle)
^{31}
32
       IN
                dest
                                           rank of destination (integer)
33
       IN
                                           message tag (integer)
                tag
34
       IN
                                           communicator (handle)
                comm
35
36
37
     C binding
38
     int MPI_Rsend(const void *buf, int count, MPI_Datatype datatype, int dest,
39
                    int tag, MPI_Comm comm)
40
     int MPI_Rsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,
41
                    int dest, int tag, MPI_Comm comm)
42
43
     Fortran 2008 binding
^{44}
     MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
46
         INTEGER, INTENT(IN) :: count, dest, tag
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
         TYPE(MPI_Comm), INTENT(IN) :: comm
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI\_Rsend(buf, count, datatype, dest, tag, comm, ierror) !(\_c)
TYPE(\*), DIMENSION(..), INTENT(IN) :: buf
INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count
TYPE(MPI\_Datatype), INTENT(IN) :: datatype
INTEGER, INTENT(IN) :: dest, tag
TYPE(MPI\_Comm), INTENT(IN) :: comm
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI\_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(\*)

```
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Send in ready mode.

There is only one receive operation, but it matches any of the send modes. The receive procedure described in the last section is *blocking*: it returns only after the receive buffer contains the newly received message. A receive can complete before the matching send has completed (of course, it can complete only after the matching send has started).

In a multithreaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. In such a case it is the user's responsibility not to modify a communication buffer until the communication completes. Otherwise, the outcome of the computation is undefined.

Advice to implementors. Since a synchronous send cannot complete before a matching receive is posted, one will not normally buffer messages sent by such an operation.

It is recommended to choose buffering over blocking the sender, whenever possible, for standard sends. The programmer can signal a preference for blocking the sender until a matching receive occurs by using the synchronous send mode.

A possible communication protocol for the various communication modes is outlined below.

ready send: The message is sent as soon as possible.

synchronous send: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

*standard send*: First protocol may be used for short messages, and second protocol for long messages.

*buffered send*: The sender copies the message into a buffer and then sends it with a nonblocking send (using the same protocol as for standard send).

Additional control messages might be needed for flow control and error recovery. Of course, there are many other possible protocols.

Ready send can be implemented as a standard send. In this case there will be no performance advantage (or disadvantage) for the use of ready send.

A standard send can be implemented as a synchronous send. In such a case, no data buffering is needed. However, users may expect some buffering.

 $\frac{24}{25}$ 

 $^{31}$ 

 $46 \\ 47$ 

In a multithreaded environment, the execution of a blocking communication should block only the executing thread, allowing the thread scheduler to de-schedule this thread and schedule another thread for execution. (End of advice to implementors.)

#### 3.5 Semantics of Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

10 Order Messages are **non-overtaking**: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI\_ANY\_SOURCE is not used in receives. (Some of the calls described later, such as MPI\_CANCEL or MPI\_WAITANY, are additional sources of nondeterminism.)

19If a process has a single thread of execution, then any two communications executed 20by this process are **ordered**. On the other hand, if the process is multithreaded, then the 21semantics of thread execution may not define a relative order between two send operations 22 executed by two distinct threads. The operations are **logically concurrent**, even if one 23physically precedes the other. In such a case, the two messages sent can be received in  $^{24}$ any order. Similarly, if two receive operations that are **logically concurrent** receive two 25successively sent messages, then the two messages can match the two receives in either 26order. 27

```
Example 3.6 An example of non-overtaking messages.
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
   CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
   CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

42**Progress** If a pair of matching send and receives have been initiated on two processes, then 43at least one of these two operations will complete, independently of other actions in the  $^{44}$ system: the send operation will complete, unless the receive is satisfied by another message, 45and completes; the receive operation will complete, unless the message sent is consumed by 46another matching receive that was posted at the same destination process.

1

 $\mathbf{2}$ 

3

4 5

> 6  $\overline{7}$

8

9

11

12

13

14

15

16

17

18

28

29

30

 $^{31}$ 

32

33 34

35

36

37 38

39

40 41

**Example 3.7** An example of two, intertwined matching pairs.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

Fairness MPI makes no guarantee of fairness in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multithreaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations.

Resource limitations Any pending communication operation consumes system resources that are limited. Errors may occur when lack of resources prevent the execution of an MPI call. A quality implementation will use a (small) fixed amount of resources for each pending send in the ready or synchronous mode and for each pending receive. However, buffer space may be consumed to store messages sent in standard mode, and must be consumed to store messages sent in buffered mode, when no matching receive is available. The amount of space available for buffering will be much smaller than program data memory on many systems. Then, it will be easy to write programs that overrun available buffer space.

MPI allows the user to provide buffer memory for messages sent in the buffered mode. Furthermore, MPI specifies a detailed operational model for the use of this buffer. An MPI implementation is required to do no worse than implied by this model. This allows users to avoid buffer overflows when they use buffered sends. Buffer allocation and use is described in Section 3.6.

A buffered send operation that cannot complete because of a lack of buffer space is *erroneous.* When such a situation is detected, an error is signaled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values 

 $\mathbf{2}$ 

 $^{24}$ 

 $^{31}$ 

and sends them to a consumer. Assume that the producer produces new values faster
 than the consumer can consume them. If buffered sends are used, then a buffer overflow
 will result. Additional synchronization has to be added to the program so as to prevent
 this from occurring. If standard sends are used, then the producer will be automatically
 throttled, as its send operations will block when buffer space is unavailable.

In some situations, a lack of buffer space leads to deadlock situations. This is illustrated by the examples below.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

This program will succeed even if no buffer space for data is available. The standard send operation can be replaced, in this example, with a synchronous send.

**Example 3.9** An errant attempt to exchange messages.

**Example 3.8** An exchange of messages.

```
! ----- THIS EXAMPLE IS ERRONEOUS -----
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

The receive operation of the first process must complete before its send, and can complete only if the matching send of the second processor is executed. The receive operation of the second process must complete before its send and can complete only if the matching send of the first process is executed. This program will always deadlock. The same holds for any other send mode.

39 40 41

42

43

44

45

46

47

48

**Example 3.10** An exchange that relies on buffering.

```
! ----- THIS EXAMPLE IS ERRONEOUS -----
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank .EQ. 1) THEN
```

6

7

8

9 10

11

12

13

14

15

16

17

18 19

20

21 22

23 24

25

26

27

28

29

30

 $^{31}$ 

32

33 34

35

36

37

```
CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by each process has to be copied out before the send operation returns and the receive operation starts. For the program to complete, it is necessary that at least one of the two messages sent be buffered. Thus, this program can succeed only if the communication system can buffer at least **count** words of data.

Advice to users. When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.

A program is "safe" if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used.

Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 3.10. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will not *deadlock*. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

Nonblocking message-passing operations, as described in Section 3.7, can be used to avoid the need for buffering outgoing messages. This prevents deadlocks due to lack of buffer space, and improves performance, by allowing overlap of computation and communication, and avoiding the overheads of allocating buffers and copying messages into buffers. (*End of advice to users.*)

# 3.6 Buffer Allocation and Usage

A user may specify a buffer to be used for buffering messages sent in buffered mode. Buffering is done by the sender.

MPI\_BUFFER\_ATTACH(buffer, size)

IN	buffer	initial buffer address (choice)
IN	size	buffer size, in bytes (non-negative integer)

C binding

```
int MPI_Buffer_attach(void *buffer, int size)
```

 $\mathbf{2}$ 

3

4

5

6 7

8 9

10

11 12

13

14 15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30 31

32 33

34 35 36

37 38

39

40 41 42

47

```
1
     int MPI_Buffer_attach_c(void *buffer, MPI_Count size)
\mathbf{2}
     Fortran 2008 binding
3
     MPI_Buffer_attach(buffer, size, ierror)
4
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
5
         INTEGER, INTENT(IN) :: size
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Buffer_attach(buffer, size, ierror) !(_c)
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
10
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     Fortran binding
13
     MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
14
          <type> BUFFER(*)
15
         INTEGER SIZE, IERROR
16
17
         Provides to MPI a buffer in the user's memory to be used for buffering outgoing mes-
18
     sages. The buffer is used only by messages sent in buffered mode. Only one buffer can be
19
     attached to a process at a time. In C, buffer is the starting address of a memory region. In
20
     Fortran, one can pass the first element of a memory region or a whole array, which must be
21
     'simply contiguous' (for 'simply contiguous,' see also Section 19.1.12).
22
23
     MPI_BUFFER_DETACH(buffer_addr, size)
^{24}
25
       OUT
                buffer_addr
                                            initial buffer address (choice)
26
       OUT
                size
                                            buffer size, in bytes (integer)
27
28
     C binding
29
     int MPI_Buffer_detach(void *buffer_addr, int *size)
30
^{31}
     int MPI_Buffer_detach_c(void *buffer_addr, MPI_Count *size)
32
     Fortran 2008 binding
33
     MPI_Buffer_detach(buffer_addr, size, ierror)
34
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
35
         TYPE(C_PTR), INTENT(OUT) :: buffer_addr
36
         INTEGER, INTENT(OUT) :: size
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Buffer_detach(buffer_addr, size, ierror) !(_c)
40
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
41
         TYPE(C_PTR), INTENT(OUT) :: buffer_addr
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     Fortran binding
45
     MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
46
47
          <type> BUFFER_ADDR(*)
         INTEGER SIZE, IERROR
48
```

Detach the buffer currently associated with MPI. The call returns the address and the size of the detached buffer. This procedure will block until all messages currently in the buffer have been transmitted. Upon return of this function, the user may reuse or deallocate the space taken by the buffer.

If the size of the detached buffer cannot be represented in size, it is set to MPI\_UNDEFINED.

```
Example 3.11 Calls to attach and detach buffers.
#define BUFFSIZE 10000
int size;
char *buff;
MPI_Buffer_attach(malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach(&buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach(buff, size);
/* Buffer of 10000 bytes available again */
```

Advice to users. Even though the C functions MPI\_Buffer\_attach and MPI\_Buffer\_detach both have a first argument of type void\*, these arguments are used differently: A pointer to the buffer is passed to MPI\_Buffer\_attach; the address of the pointer is passed to MPI\_Buffer\_detach, so that this call can return the pointer value. In Fortran with the mpi module or mpif.h, the type of the buffer\_addr argument is wrongly defined and the argument is therefore unused. In Fortran with the mpi\_f08 module, the address of the buffer is returned as TYPE(C\_PTR), see also Example 9.1 about the use of C\_PTR pointers. (*End of advice to users.*)

Rationale. Both arguments are defined to be of type void\* (rather than void\* and void\*\*, respectively), so as to avoid complex type casts. E.g., in the last example, &buff, which is of type char\*\*, can be passed as argument to MPI\_Buffer\_detach without type casting. If the formal parameter had type void\*\* then we would need a type cast before and after the call. (*End of rationale.*)

The statements made in this section describe the behavior of MPI for buffered-mode sends. When no buffer is currently associated, MPI behaves as if a zero-sized buffer is associated with the process.

MPI must provide as much buffering for outgoing messages *as if* outgoing message data were buffered by the sending process, in the specified buffer space, using a circular, contiguous-space allocation policy. We outline below a model implementation that defines this policy. MPI may provide more buffering, and may use a better buffer allocation algorithm than described below. On the other hand, MPI may signal an error whenever the simple buffering allocator described below would run out of space. In particular, if no buffer is explicitly associated with the process, then any buffered send may cause an error.

MPI does not provide mechanisms for querying or controlling buffering done by standard mode sends. It is expected that vendors will provide such information for their implementations.

*Rationale.* There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be

 $^{31}$ 

dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (*End of rationale.*)

7 8 9

17 18

21

22 23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32 33

34

35 36

37

1

2

3

4

5

6

# 3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 5.2 and the nonblocking communication functions described in Section 3.7.

We assume that a circular queue of pending message entries (PME) is maintained. Each entry contains a communication request handle that identifies a pending nonblocking send, a pointer to the next entry and the packed message data. The entries are stored in successive locations in the buffer. Free space is available between the queue tail and the queue head.

A buffered send call results in the execution of the following code.

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
  - Compute the number, n, of bytes needed to store an entry for the new message. An upper bound on n can be computed as follows: A call to the function MPI\_PACK\_SIZE(count, datatype, comm, size), with the count, datatype and comm arguments used in the MPI\_BSEND call, returns an upper bound on the amount
    - of space needed to buffer the message data (see Section 5.2). The MPI constant MPI\_BSEND\_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or *envelope* information).
    - Find the next contiguous empty space of n bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
      - Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI\_PACK is used to pack data.
      - Post nonblocking send (standard mode) for packed data.

• Return

38 39 40

41

3.7 Nonblocking Communication

<sup>42</sup> Nonblocking communication is important both for reasons of correctness and perfor-<sup>43</sup>mance. For complex communication patterns, the use of only blocking communication <sup>44</sup> (without buffering) is difficult because the programmer must ensure that each send is <sup>45</sup>matched with a receive in an order that avoids *deadlock*. For communication patterns that <sup>46</sup>are determined only at run time, this is even more difficult. Nonblocking communication <sup>47</sup>can be used to avoid this problem, allowing programmers to express complex and possibly <sup>48</sup>dynamic communication patterns without needing to ensure that all sends and receives are issued in an order that prevents deadlock (see Section 3.5 and the discussion of "safe" programs). Nonblocking communication also allows for the *overlap* of communication with different communication operations, e.g., to prevent the *serialization* of such operations, and for the *overlap* of communication with computation. Whether an implementation is able to accomplish an effective (from a performance standpoint) overlap of operations depends on the implementation itself and the system on which the implementation is running. Using nonblocking operations *permits* an implementation to overlap communication with computation, but does not require it to do so.

9 A nonblocking **send start** call *initiates* the send operation, but does not complete it. 10 The send start call can return before the message was copied out of the send buffer. A 11separate send complete call is needed to complete the communication, i.e., to verify that 12the data has been copied out of the send buffer. With suitable hardware, the transfer of data 13 out of the sender memory may proceed concurrently with computations done at the sender 14after the send was initiated and before it completed. Similarly, a nonblocking receive start 15call *initiates* the receive operation, but does not complete it. The call can return before a 16message is stored into the receive buffer. A separate **receive complete** call is needed to 17 complete the receive operation and verify that the data has been received into the receive 18 buffer. With suitable hardware, the transfer of data into the receiver memory may proceed 19concurrently with computations done after the receive was initiated and before it completed. 20The use of nonblocking receives may also avoid system buffering and memory-to-memory 21copying, as information is provided early on the location of the receive buffer.

Nonblocking send start calls can use the same four modes as blocking sends: standard, 22buffered, synchronous, and ready. These carry the same meaning. Sends of all modes, ready 23 $^{24}$ excepted, can be started whether a matching receive has been posted or not; a nonblocking 25ready send can be started only if a matching receive is posted. In all cases, the send start 26call is *local*: it returns immediately, irrespective of the status of other processes. If the call causes some system resource to be exhausted, then it will fail and return an error code. 2728 Quality implementations of MPI should ensure that this happens only in "pathological" 29cases. That is, an MPI implementation should be able to support a large number of pending 30 nonblocking operations.

The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode.

If the send mode is **synchronous**, then the send can complete only if a matching 34receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is *non-local*. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call 36 occurs. (It can complete as soon as the sender "knows" the transfer will complete, but before the receiver "knows" the transfer will complete.)

If the send mode is **buffered** then the message must be buffered if there is no pending receive. In this case, the send-complete call is *local*, and must succeed irrespective of the status of a matching receive.

If the send mode is **standard** then the send-complete call may return before a matching receive is posted, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive is posted, and the message was copied into the receive buffer.

Nonblocking sends can be matched with blocking receives, and vice-versa.

The completion of a send operation may be delayed, for standard Advice to users.

1  $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8

31

32 33

35

37

38 39

40 41

42

43

44

4546

47

mode, and must be delayed, for synchronous mode, until a matching receive is posted. The use of nonblocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Nonblocking sends in the buffered and ready modes have a more limited impact, e.g., the blocking version of buffered send is capable of completing regardless of when a matching receive call is made. However, separating the start from the completion of these sends still gives some opportunity for optimization within the MPI library. For example, starting a buffered send gives an implementation more flexibility in determining if and how the message is buffered. There are also advantages for both nonblocking buffered and ready modes when data copying can be done concurrently with computation.

The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of nonblocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. (*End of advice to users.*)

3.7.1 Communication Request Objects

 Nonblocking communications use opaque request objects to identify communication operations and match the operation that initiates the communication with the operation that terminates it. These are system objects that are accessed via a handle. A request object identifies various properties of a communication operation, such as the send mode, the communication buffer that is associated with it, its context, the tag and destination arguments to be used for a send, or the tag and source arguments to be used for a receive. In addition, this object stores information about the status of the pending communication operation.

3.7.2 Communication Initiation

For the functions defined in this section, we use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for *buffered*, *synchronous*, or *ready* mode. In addition, for these functions a prefix of I (for *immediate* and *incomplete*) indicates that the call is nonblocking.

37

 $^{31}$ 

32

- 38
- 39 40
- 41
- 42

43

- 44
- 45
- 46
- 47
- 48

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15 16 17

18

19

20 21

MPI_ISEND(buf, count, datatype, dest, tag, comm, request) <sup>1</sup>				
IN	buf	initial address of send buffer (choice)	2	
IN	count	number of elements in send buffer (non-negative	3 4	
		integer)	5	
IN	datatype	datatype of each send buffer element (handle)	6	
IN	dest	rank of destination (integer)	7 8	
IN	tag	message tag (integer)	9	
IN	comm	communicator (handle)	10	
OU	T request	communication request (handle)	11	
		- 、 ,	12 13	
	nding		14	
int		*buf, int count, MPI_Datatype datatype, int dest,	15	
	int tag, MP	_Comm comm, MPI_Request *request)	16	
int		d *buf, MPI_Count count, MPI_Datatype datatype,	17 18	
	int dest, in	t tag, MPI_Comm comm, MPI_Request *request)	19	
	ran 2008 binding		20	
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)				
		), INTENT(IN), ASYNCHRONOUS :: buf	22	
	INTEGER, INTENT(IN) :: count, dest, tag			
	TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Comm), INTENT(IN) :: comm			
	TYPE(MPI_Comm), INTEN TYPE(MPI_Request), IN		25	
	INTEGER, OPTIONAL, IN	-	26	
			27 28	
		atype, dest, tag, comm, request, ierror) !(_c)	28 29	
		), INTENT(IN), ASYNCHRONOUS :: buf	30	
		T_KIND), INTENT(IN) :: count	31	
		NTENT(IN) :: datatype	32	
	INTEGER, INTENT(IN) :	-	33	
	TYPE(MPI_Comm), INTEN TYPE(MPI_Request), IN		34	
	INTEGER, OPTIONAL, IN	-	35	
	INIEGEN, OFFICIAL, IN		36	
Fort	ran binding		37	
	MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) <type> BUF(*)</type>			
	INTEGER COUNT, DATATY	PE, DEST, TAG, COMM, REQUEST, IERROR	40	
(	Start a standard mode no	onblocking send.	41	
		-	42	
			43	
			44	
			45 46	
			40	

# MPL ISEND(buf count datatype dest tag comm request)

1 MPI\_IBSEND(buf, count, datatype, dest, tag, comm, request)  $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN count number of elements in send buffer (non-negative 4 integer) 56 IN datatype datatype of each send buffer element (handle) 7 dest IN rank of destination (integer) 8 IN tag message tag (integer) 9 10IN comm communicator (handle) 11 OUT request communication request (handle) 1213 C binding 14 int MPI\_Ibsend(const void \*buf, int count, MPI\_Datatype datatype, int dest, 15int tag, MPI\_Comm comm, MPI\_Request \*request) 1617 int MPI\_Ibsend\_c(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 18 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 19Fortran 2008 binding 20MPI\_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) 21TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 22 INTEGER, INTENT(IN) :: count, dest, tag 23TYPE(MPI\_Datatype), INTENT(IN) :: datatype 24TYPE(MPI\_Comm), INTENT(IN) :: comm 25TYPE(MPI\_Request), INTENT(OUT) :: request 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728MPI\_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) !(\_c) 29TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 30 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 31TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 INTEGER, INTENT(IN) :: dest, tag 33 TYPE(MPI\_Comm), INTENT(IN) :: comm 34 TYPE(MPI\_Request), INTENT(OUT) :: request 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 Fortran binding 37 MPI\_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 38 <type> BUF(\*) 39 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 4041 Start a buffered mode nonblocking send. 4243 44 4546 47 48

MPI_	ISSEND(buf, count, datat	type, dest, tag, comm, request)	1
IN	buf	initial address of send buffer (choice)	2 3
IN	count	number of elements in send buffer (non-negative integer)	4 5
IN	datatype	datatype of each send buffer element (handle)	6
IN	dest	rank of destination (integer)	7
IN	tag	message tag (integer)	8 9
IN	comm	communicator (handle)	10
OU.			11
00	T request	communication request (handle)	12
C bi	nding		13
	0	*buf, int count, MPI_Datatype datatype, int dest,	14 15
	int tag, MPI	I_Comm comm, MPI_Request *request)	16
int N	PI_Issend_c(const vo	id *buf, MPI_Count count, MPI_Datatype datatype,	17
		nt tag, MPI_Comm comm, MPI_Request *request)	18
Forti	an 2008 binding		19
	0	tatype, dest, tag, comm, request, ierror)	20 21
		), INTENT(IN), ASYNCHRONOUS :: buf	22
	INTEGER, INTENT(IN) :	-	23
		NTENT(IN) :: datatype	24
	TYPE(MPI_Comm), INTEN		25
	TYPE(MPI_Request), IN INTEGER, OPTIONAL, IN	-	26
			27 28
		tatype, dest, tag, comm, request, ierror) !(_c)	28 29
		), INTENT(IN), ASYNCHRONOUS :: buf T_KIND), INTENT(IN) :: count	30
		NTENT(IN) :: datatype	31
	INTEGER, INTENT(IN) :		32
	TYPE(MPI_Comm), INTEN	-	33
1	TYPE(MPI_Request), IN	TENT(OUT) :: request	34
]	INTEGER, OPTIONAL, IN	TENT(OUT) :: ierror	35 36
Forti	ran binding		37
MPI_I	ISSEND(BUF, COUNT, DA	TATYPE, DEST, TAG, COMM, REQUEST, IERROR)	38
	<type> BUF(*)</type>		39
]	INTEGER COUNT, DATATY	PE, DEST, TAG, COMM, REQUEST, IERROR	40
S	start a synchronous mod	e nonblocking send.	41
			42
			43 44
			45
			46
			47

#### mm request) MDI ISSEND(buf . daa

1 MPI\_IRSEND(buf, count, datatype, dest, tag, comm, request)  $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN count number of elements in send buffer (non-negative 4 integer) 56 IN datatype datatype of each send buffer element (handle) 7 dest IN rank of destination (integer) 8 IN tag message tag (integer) 9 10IN comm communicator (handle) 11 OUT request communication request (handle) 1213 C binding 14 int MPI\_Irsend(const void \*buf, int count, MPI\_Datatype datatype, int dest, 15int tag, MPI\_Comm comm, MPI\_Request \*request) 1617 int MPI\_Irsend\_c(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 18 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 19Fortran 2008 binding 20MPI\_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) 21TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 22 INTEGER, INTENT(IN) :: count, dest, tag 23TYPE(MPI\_Datatype), INTENT(IN) :: datatype 24TYPE(MPI\_Comm), INTENT(IN) :: comm 25TYPE(MPI\_Request), INTENT(OUT) :: request 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728MPI\_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) !(\_c) 29TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 30 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 31TYPE(MPI\_Datatype), INTENT(IN) :: datatype 32 INTEGER, INTENT(IN) :: dest, tag 33 TYPE(MPI\_Comm), INTENT(IN) :: comm 34 TYPE(MPI\_Request), INTENT(OUT) :: request 35INTEGER, OPTIONAL, INTENT(OUT) :: ierror 36 Fortran binding 37 MPI\_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 38 <type> BUF(\*) 39 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 4041 Start a ready mode nonblocking send. 4243 44 4546 47 48

MPI_	IRECV(buf, count, datatyp	be, source, tag, comm, request)	1
OU	T buf	initial address of receive buffer (choice)	2 3
IN	count	number of elements in receive buffer (non-negative integer)	4 5
IN	datatype	datatype of each receive buffer element (handle)	6
IN	source	rank of source or MPI_ANY_SOURCE (integer)	7
IN	tag	message tag or MPI_ANY_TAG (integer)	8 9
IN	comm	communicator (handle)	10
OU		communication request (handle)	11 12
		int count, MPI_Datatype datatype, int source, _Comm comm, MPI_Request *request)	13 14 15 16
int		, MPI_Count count, MPI_Datatype datatype, int tag, MPI_Comm comm, MPI_Request *request)	17 18 19
MPI_		NTENT(IN) :: datatype F(IN) :: comm FENT(OUT) :: request	20 21 22 23 24 25 26 27
	TYPE(*), DIMENSION()	F_KIND), INTENT(IN) :: count NTENT(IN) :: datatype : source, tag F(IN) :: comm FENT(OUT) :: request	28 29 30 31 32 33 34 35 26
MPI_	<type> BUF(*)</type>	ATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) PE, SOURCE, TAG, COMM, REQUEST, IERROR	36 37 38 39 40
	Start a nonblocking receiv		41 42 43 44 45 46

12	MPI_ISEN	DRECV(sendbuf, sendcount, s source, recvtag, comm,	endtype, dest, sendtag, recvbuf, recvcount, recvtype, request)
3 4	IN	sendbuf	initial address of send buffer (choice)
5 6	IN	sendcount	number of elements in send buffer (non-negative integer)
7	IN	sendtype	datatype of each send buffer element (handle)
8 9	IN	dest	rank of destination (integer)
9 10	IN	sendtag	send tag (integer)
11	OUT	recvbuf	initial address of receive buffer (choice)
12 13 14	IN	recvcount	number of elements in receive buffer (non-negative integer)
15	IN	recvtype	datatype of each receive buffer element (handle)
16	IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)
17	IN	recvtag	receive tag or $MPI_ANY_TAG$ (integer)
18 19	IN	comm	communicator (handle)
20	OUT	request	communication request (handle)
21 22			
23 24 25 26 27	<pre>int MPI_Isendrecv(const void *sendbur, int sendcount,</pre>		
28 29 30 31	int MPI_]	MPI_Datatype sendty MPI_Count recvcount	sendbuf, MPI_Count sendcount, pe, int dest, int sendtag, void *recvbuf, , MPI_Datatype recvtype, int source, mm comm, MPI_Request *request)
32	Fortran 2	2008 binding	
33 34	MPI_Isend		, sendtype, dest, sendtag, recvbuf,
35	recvcount, recvtype, source, recvtag, comm, request, ierror)		
36	<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,</pre>		
37	recvtag		
38 39	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype		
40	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf		
41	TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Request), INTENT(OUT) :: request		
42		GER, OPTIONAL, INTENT(OUT	-
43			
44 $45$	MP1_1send		, sendtype, dest, sendtag, recvbuf, , source, recvtag, comm, request, ierror)
46		!(_c)	, searce, recevery, comm, request, rerier,
47	TYPE		NT(IN), ASYNCHRONOUS :: sendbuf
48	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount		

	TYPE(M	PI_Datatype), INTENT(IN)	:: sendtype, recvtype	1
		R, INTENT(IN) :: dest, s	5	2
		), DIMENSION(), ASYNCH		3
		PI_Comm), INTENT(IN) ::		4
		PI_Request), INTENT(OUT) R, OPTIONAL, INTENT(OUT)	-	5 6
	INIEGE	R, UPIIONAL, INIENI(UUI)	:: Terror	7
	ran bi	0		8
MPI_	ISENDR		SENDTYPE, DEST, SENDTAG, RECVBUF,	9
	<+	RECVCUUNT, RECVTYPE, SENDBUF(*), RECVBUF(*)	SOURCE, RECVTAG, COMM, REQUEST, IERROR)	10
			EST, SENDTAG, RECVCOUNT, RECVTYPE,	11
	1111202	SOURCE, RECVTAG, COM		12 13
	Tuitiata			13
	Initiate	a honolocking communicatio	n request for a <i>send and receive</i> operation.	15
				16
MPI.	_ISEND		datatype, dest, sendtag, source, recvtag, comm,	17
		request)		18
IN	OUT	buf	initial address of send and receive buffer (choice)	19 20
IN		count	number of elements in send and receive buffer	20
			(non-negative integer)	22
IN		datatype	type of elements in send and receive buffer (handle)	23
IN		dest	rank of destination (integer)	24
IN		sendtag	send message tag (integer)	25 26
IN		source	rank of source or $MPI\_ANY\_SOURCE$ (integer)	27
IN		recvtag	receive message tag or $MPI\_ANY\_TAG$ (integer)	28
IN		comm	communicator (handle)	29 30
οι	JT	request	communication request (handle)	31
				32
C b	inding			33
int	MPI_Is	endrecv_replace(void *bu	f, int count, MPI_Datatype datatype,	34
		•	, int source, int recvtag, MPI_Comm comm,	35
		MPI_Request *request)		36 37
int	MPI_Is	endrecv_replace_c(void *	buf, MPI_Count count,	38
			, int dest, int sendtag, int source,	39
		int recvtag, MPI_Comm	comm, MPI_Request *request)	40
Fort	ran 20	08 binding		41
MPI_	Isendr	ecv_replace(buf, count,	datatype, dest, sendtag, source, recvtag,	42
		comm, request, ierror		43
		), DIMENSION(), ASYNCH		44 45
			dest, sendtag, source, recvtag	40 46
		PI_Datatype), INTENT(IN) PI_Comm), INTENT(IN) :: •		47
		PI_Request), INTENT(IN)		48

INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Isendrecv\_replace(buf, count, datatype, dest, sendtag, source, recvtag, comm, request, ierror) !(\_c) TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: buf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI\_ISENDRECV\_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST, IERROR) <type> BUF(\*) INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST, IERROR Initiate a nonblocking communication request for a send and receive operation. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received. These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion. A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes. A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes. Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10-19.1.20. (End of advice to users.)

34 35 36

# 3.7.3 Communication Completion

The functions MPI\_WAIT and MPI\_TEST are used to complete a nonblocking communication. The *completion* of a send operation indicates that the sender is now free to update the locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the message has been received, rather, it may have been buffered by the communication subsystem. However, if a *synchronous mode send* was used, the *completion* of the send operation indicates that a matching receive was *initiated*, and that the message will eventually be received by this matching receive.

The *completion* of a receive operation indicates that the receive buffer contains the received message, the receiver is now free to access it, and that the status object is set. It does not indicate that the matching send operation has *completed* (but indicates, of course, that the send was *initiated*).

1

2

3

4

5

6

7

8

9

10

11 12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30 31

32

We shall use the following terminology: A **null handle** is a handle with value MPI\_REQUEST\_NULL. A *persistent communication request* and the handle to it are **inactive** if the request is not associated with any ongoing communication (see Section 3.9). A handle is **active** if it is neither *null* nor *inactive*. An **empty** status is a status which is set to return tag = MPI\_ANY\_TAG, source = MPI\_ANY\_SOURCE, error = MPI\_SUCCESS, and is also internally configured so that calls to MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and MPI\_GET\_ELEMENTS\_X return count = 0 and MPI\_TEST\_CANCELLED returns false. We set a status variable to *empty* when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI\_WAIT, MPI\_TEST, or any of the other derived functions (MPI\_{TEST|WAIT}{ALL|SOME|ANY}), where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI\_ERR\_IN\_STATUS; and the returned status can be queried by the call MPI\_TEST\_CANCELLED.

Error codes belonging to the error class MPI\_ERR\_IN\_STATUS should be returned only by the MPI completion functions that take arrays of MPI\_Status. For the functions that take a single MPI\_Status argument, the error code is returned by the function, and the value of the MPI\_ERROR field in the MPI\_Status argument is undefined (see 3.2.5).

			20
MPI_WAI	T(request, status)		21
INOUT	request	request (handle)	22
OUT	status	status object (status)	23
001	512115	status object (status)	24
Chindia			25
C bindir	0		26
int MPL_	Wait(MP1_Request	*request, MPI_Status *status)	27
Fortran	2008 binding		28
MPI_Wait	(request, status,	ierror)	29
TYPE	(MPI_Request), IN	TENT(INOUT) :: request	30
TYPE	(MPI_Status) :: s	tatus	31
INTE	GER, OPTIONAL, IN	TENT(OUT) :: ierror	32
_			33
Fortran	0		34
	(REQUEST, STATUS,		35
INTE	GER REQUEST, STAT	US(MPI_STATUS_SIZE), IERROR	36
			37

A call to MPI\_WAIT returns when the operation identified by request is *complete*. If the request is an *active persistent communication request*, it is marked *inactive*. Any other type of request is deallocated and the request handle is set to MPI\_REQUEST\_NULL. MPI\_WAIT is a *non-local* procedure.

The call returns, in status, information on the completed operation. The content of the status object for a receive operation can be accessed as described in Section 3.2.5. The status object for a send operation may be queried by a call to MPI\_TEST\_CANCELLED (see Section 3.8).

One is allowed to call MPI\_WAIT with a *null* or *inactive* request argument. In this case the procedure returns immediately with *empty* status.

Advice to users. Successful return of MPI\_WAIT after a MPI\_IBSEND implies that

 $\mathbf{2}$ 

1 the user send buffer can be reused—i.e., data has been sent out or copied into a buffer 2 attached with MPI\_BUFFER\_ATTACH. Note that, at this point, we can no longer 3 cancel the send (see Section 3.8). If a matching receive is never posted, then the 4 buffer cannot be freed. This runs somewhat counter to the stated goal of MPI\_CANCEL 5(always being able to free program space that was committed to the communication 6 subsystem). (End of advice to users.) 7 Advice to implementors. In a multithreaded environment, a call to MPI\_WAIT should 8 block only the calling thread, allowing the thread scheduler to schedule another thread 9 for execution. (End of advice to implementors.) 10 11 1213MPI\_TEST(request, flag, status) 14INOUT request communication request (handle) 1516OUT true if operation completed (logical) flag 17 OUT status status object (status) 18 19C binding 20int MPI\_Test(MPI\_Request \*request, int \*flag, MPI\_Status \*status) 2122 Fortran 2008 binding 23MPI\_Test(request, flag, status, ierror)  $^{24}$ TYPE(MPI\_Request), INTENT(INOUT) :: request 25LOGICAL, INTENT(OUT) :: flag 26TYPE(MPI\_Status) :: status 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 28Fortran binding 29 MPI\_TEST(REQUEST, FLAG, STATUS, IERROR) 30 INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR  $^{31}$ LOGICAL FLAG 32 33 A call to MPI\_TEST returns flag = true if the operation identified by request is *complete*. 34In such a case, the status object is set to contain information on the completed operation. 35 If the request is an active persistent communication request, it is marked as inactive. Any 36 other type of request is deallocated and the request handle is set to MPI\_REQUEST\_NULL. 37 The call returns flag = false if the operation identified by request is not complete. In this 38 case, the value of the status object is undefined. MPI\_TEST is a *local* procedure. 39 The return status object for a receive operation carries information that can be accessed 40as described in Section 3.2.5. The status object for a send operation carries information 41 that can be accessed by a call to MPI\_TEST\_CANCELLED (see Section 3.8). 42One is allowed to call MPI\_TEST with a *null* or *inactive* request argument. In such a 43 case the procedure returns with flag = true and *empty* status. 44The procedures MPI\_WAIT and MPI\_TEST can be used to complete any request-based 45nonblocking or persistent operation. 4647The use of the nonblocking MPI\_TEST call allows the user to Advice to users. 48

schedule alternative activities within a single thread of execution. An event-driven

thread scheduler can be emulated with periodic calls to MPI\_TEST. (*End of advice to users.*)

	- 1
<b>Example 3.12</b> Simple usage of nonblocking operations and MPI_WAIT.	4 5
CALL MPI_COMM_RANK(comm, rank, ierr) IF (rank .EQ. 0) THEN CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)	6 7 8
**** do some computation to mask latency **** CALL MPI_WAIT(request, status, ierr) ELSE IF (rank .EQ. 1) THEN	9 10 11
CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr) **** do some computation to mask latency ****	12 13 14
CALL MPI_WAIT(request, status, ierr) END IF	15
A request object can be <i>freed</i> using the following MPI procedure.	16 17
	18
MPI_REQUEST_FREE(request)	19 20
INOUT request (handle)	21
	22
C binding int MPI_Request_free(MPI_Request *request)	23 24
Fortran 2008 binding	25
MPI_Request_free(request, ierror)	26
TYPE(MPI_Request), INTENT(INOUT) :: request	27 28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
Fortran binding	30
MPI_REQUEST_FREE(REQUEST, IERROR) INTEGER REQUEST, IERROR	31 32
	33
MPI_REQUEST_FREE is a <i>local</i> procedure. Upon successful return, MPI_REQUEST_FREE sets request to MPI_REQUEST_NULL. For an <i>inactive</i>	34
request representing any type of MPI operation, MPI_REQUEST_FREE shall do the <i>freeing</i>	35 36
stage of the associated operation during its execution.	30
For a request representing a <i>nonblocking</i> point-to-point or a persistent point-to-point	38
operation, it is permitted (although strongly discouraged) to call MPI_REQUEST_FREE when the request is <i>active</i> . In this special case, MPI_REQUEST_FREE will only mark the	39
request for freeing and MPI will actually do the <i>freeing stage</i> of the associated operation	40 41
later.	41
The use of this procedure for generalized requests is described in Section 13.2.	43
Calling MPI_REQUEST_FREE with an <i>active</i> request representing any other type of MPI operation (e.g., any partitioned operation (see Chapter 4), any collective operation	44
(see Chapter 6), any I/O operation (see Chapter 14), or any request-based RMA operation	45 46
(see Chapter 12)) is <i>erroneous</i> .	47

2

*Rationale.* For point-to-point operations, the MPI\_REQUEST\_FREE mechanism is provided for reasons of performance and convenience on the sending side. (*End of rationale.*)

Advice to users. Once a request is freed by a call to MPI\_REQUEST\_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI\_WAIT or MPI\_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user—such an error must be treated as fatal. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (*End of advice to users.*)

```
Example 3.13 An example using MPI_REQUEST_FREE.
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank .EQ. 0) THEN
   DO i=1,n
      CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
   END DO
ELSE IF (rank .EQ. 1) THEN
   CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
   CALL MPI_WAIT(req, status, ierr)
   DO I=1,n-1
      CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_REQUEST_FREE(req, ierr)
      CALL MPI_IRECV(inval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
      CALL MPI_WAIT(req, status, ierr)
   END DO
   CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
   CALL MPI_WAIT(req, status, ierr)
END IF
```

### 3.7.4 Semantics of Nonblocking Communications

The semantics of nonblocking communication is defined by suitably extending the definitions in Section 3.5.

Order Nonblocking communication operations are ordered according to the execution order of the calls that *initiate* the communication. The **non-overtaking** requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used.

```
Example 3.14 Message ordering for nonblocking operations.
```

```
CALL MPI_COMM_RANK(comm, rank, ierr)
```

```
<sup>47</sup>
<sup>48</sup> IF (RANK .EQ. 0) THEN
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11 12

13 14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34 35

36 37

38

39 40

41

42

43

44

```
CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)

CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)

ELSE IF (rank .EQ. 1) THEN

CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)

CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)

END IF

CALL MPI_WAIT(r1, status, ierr)

CALL MPI_WAIT(r2, status, ierr)
```

The first send of process zero will match the first receive of process one, even if both messages are sent before process one executes either receive.

**Progress** A call to MPI\_WAIT that *completes* a receive will eventually terminate and return if a matching send has been *started*, unless the send is satisfied by another receive. In particular, if the matching send is *nonblocking*, then the receive should *complete* even if no call is executed by the sender to *complete* the send. Similarly, a call to MPI\_WAIT that *completes* a send will eventually return if a matching receive has been *started*, unless the receive is satisfied by another send, and even if no call is executed to *complete* the receive.

```
Example 3.15 An illustration of progress semantics.
```

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK .EQ. 0) THEN
CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, status, ierr)
CALL MPI_WAIT(r, status, ierr)
END IF
```

This code should not deadlock in a correct MPI implementation. The first synchronous send of process zero must complete after process one posts the matching (nonblocking) receive even if process one has not yet reached the completing wait call. Thus, process zero will continue and execute the second send, allowing process one to complete execution.

If an MPI\_TEST that *completes* a receive is repeatedly called with the same arguments, and a matching send has been *started*, then the call will eventually return flag = true, unless the send is satisfied by another receive. If an MPI\_TEST that *completes* a send is repeatedly called with the same arguments, and a matching receive has been *started*, then the call will eventually return flag = true, unless the receive is satisfied by another send.

# 3.7.5 Multiple Completions

It is convenient to be able to wait for the *completion* of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI\_WAITANY or MPI\_TESTANY can be used to wait for the *completion* of one out of several operations. A call to MPI\_WAITALL or MPI\_TESTALL can be used to wait for all pending operations in a list. A call to MPI\_WAITSOME or MPI\_TESTSOME can be used to *complete* all enabled

```
1
      operations in a list.
\mathbf{2}
3
      MPI_WAITANY(count, array_of_requests, index, status)
4
5
       IN
                 count
                                              list length (non-negative integer)
6
       INOUT
                 array_of_requests
                                              array of requests (array of handles)
7
       OUT
                 index
                                              index of handle for operation that completed
8
                                              (integer)
9
10
        OUT
                 status
                                              status object (status)
11
12
      C binding
13
      int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,
14
                     MPI_Status *status)
15
      Fortran 2008 binding
16
     MPI_Waitany(count, array_of_requests, index, status, ierror)
17
          INTEGER, INTENT(IN) :: count
18
          TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
19
          INTEGER, INTENT(OUT) :: index
20
          TYPE(MPI_Status) :: status
21
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     Fortran binding
24
     MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
25
          INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
26
                      IERROR
27
          Blocks until one of the operations associated with the active requests in the array has
28
      completed. If more than one operation is enabled and can terminate, one is arbitrarily
29
      chosen. Returns in index the index of that request in the array and returns in
30
      status the status of the completing operation. (The array is indexed from zero in C, and
^{31}
      from one in Fortran.) If the request is an active persistent communication request, it is
32
      marked inactive. Any other type of request is deallocated and the request handle is set to
33
34
      MPI_REQUEST_NULL.
          The array_of_requests list may contain null or inactive handles. If the list contains no
35
      active handles (list has length zero or all entries are null or inactive), then the call returns
36
      immediately with index = MPI_UNDEFINED, and an empty status.
37
          The execution of MPI_WAITANY with an array containing multiple entries has the
38
      same effect as the execution of MPI_WAIT with the array entry indicated by the output
39
      value of index (unless the output value of index is MPI_UNDEFINED). MPI_WAITANY with
40
      an array containing one active entry is equivalent to MPI_WAIT.
41
42
43
44
45
46
47
48
```

	· · ·	- ,	
IN	count	list length (non-negative integer)	
INOUT	array_of_requests	array of requests (array of handles)	4
OUT	index	index of operation that completed or MPI_UNDEFINED if none completed (integer)	;
OUT	flag	true if one of the operations is complete (logical)	7
OUT	status	status object (status)	ę

#### MPI\_TESTANY(count, array\_of\_requests, index, flag, status)

#### C binding

#### Fortran 2008 binding

MPI\_Testany(count, array\_of\_requests, index, flag, status, ierror)
 INTEGER, INTENT(IN) :: count
 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count)
 INTEGER, INTENT(OUT) :: index
 LOGICAL, INTENT(OUT) :: flag
 TYPE(MPI\_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

```
MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
IERROR
LOGICAL FLAG
```

Tests for *completion* of either one or none of the operations associated with *active* handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation. If the request is an *active persistent communication request*, it is marked as *inactive*. Any other type of request is deallocated and the handle is set to MPI\_REQUEST\_NULL. (The array is indexed from zero in C, and from one in Fortran.) In the latter case (no operation *completed*), it returns flag = false, returns a value of MPI\_UNDEFINED in index and status is undefined.

The array may contain *null* or inactive handles. If the array contains no *active* handles then the call returns *immediately* with flag = true,  $index = MPI_UNDEFINED$ , and an *empty* status.

If the array of requests contains *active* handles then the execution of MPI\_TESTANY has the same effect as the execution of MPI\_TEST with each of the array elements in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to indicate which array element returned flag = true and in the latter case, it is set to MPI\_UNDEFINED. MPI\_TESTANY with an array containing one *active* entry is equivalent to MPI\_TEST.

 $^{24}$ 

1 MPI\_WAITALL(count, array\_of\_requests, array\_of\_statuses) 2 IN count list length (non-negative integer) 3 INOUT array\_of\_requests array of requests (array of handles) 45OUT array\_of\_statuses array of status objects (array of status) 6  $\overline{7}$ C binding 8 int MPI\_Waitall(int count, MPI\_Request array\_of\_requests[], 9 MPI\_Status array\_of\_statuses[]) 10 Fortran 2008 binding 11MPI\_Waitall(count, array\_of\_requests, array\_of\_statuses, ierror) 12INTEGER, INTENT(IN) :: count 13 TYPE(MPI\_Request), INTENT(INOUT) :: array\_of\_requests(count) 14TYPE(MPI\_Status) :: array\_of\_statuses(\*) 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 Fortran binding 18 MPI\_WAITALL(COUNT, ARRAY\_OF\_REQUESTS, ARRAY\_OF\_STATUSES, IERROR) 19INTEGER COUNT, ARRAY\_OF\_REQUESTS(\*), 20ARRAY\_OF\_STATUSES(MPI\_STATUS\_SIZE, \*), IERROR 21Blocks until all communication operations associated with *active* handles in the list 22 complete, and returns the status of all these operations (this includes the case where no 23handle in the list is *active*). Both arrays have the same number of valid entries. The  $^{24}$ i-th entry in array\_of\_statuses is set to the return status of the i-th operation. Active 25*persistent requests* are marked *inactive*. Requests of any other type are deallocated and the 26corresponding handles in the array are set to MPI\_REQUEST\_NULL. The list may contain 27null or *inactive* handles. The call sets to *empty* the status of each such entry. 28The error-free execution of MPI\_WAITALL has the same effect as the execution of 29 MPI\_WAIT for each of the array elements in some arbitrary order. MPI\_WAITALL with an 30 array of length one is equivalent to MPI\_WAIT.  $^{31}$ When one or more of the communications *completed* by a call to MPI\_WAITALL fail, 32 it is desirable to return specific information on each communication. The function 33 34MPI\_WAITALL will return in such case the error code MPI\_ERR\_IN\_STATUS and will set the error field of each status to a specific error code. This code will be MPI\_SUCCESS, if the 35 specific communication *completed*; it will be another specific error code, if it failed; or it can 36 be MPI\_ERR\_PENDING if it has neither failed nor *completed*. The function MPI\_WAITALL 37 will return MPI\_SUCCESS if no request had an error, or will return another error code if it 38 failed for other reasons (such as invalid arguments). In such cases, it will not update the 39 error fields of the statuses. 4041 Rationale. This design streamlines error handling in the application. The application 42code need only test the (single) function result to determine if an error has occurred. It 43 needs to check each individual status only when an error occurred. (End of rationale.) 44454647 48

MPI_TEST	MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)		1
IN	count	list length (non-negative integer)	2
INOUT	array_of_requests	array of requests (array of handles)	3
		* <u>-</u> ( * ,	4
OUT	flag	true if all of the operations are complete (logical)	5
OUT	array_of_statuses	array of status objects (array of status)	6 7
			8
C binding	r 0		9
int MPI_7	Cestall(int count, MPI_Re	<pre>quest array_of_requests[], int *flag,</pre>	10
	MPI_Status array_of_	statuses[])	11
Fortran 2	2008 binding		12
	0	sts, flag, array_of_statuses, ierror)	13
	ER, INTENT(IN) :: count	505, 114 <u>5</u> , 4114 <u>, 01</u> _50404505, 101101,	14
		UT) :: array_of_requests(count)	15
	CAL, INTENT(OUT) :: flag	<b>5 - 1 · · ·</b>	16
	(MPI_Status) :: array_of_	<pre>statuses(*)</pre>	17
•		18	
			19
Fortran b	0		20
		STS, FLAG, ARRAY_OF_STATUSES, IERROR)	21
INTEG	GER COUNT, ARRAY_OF_REQUE		22
		PI_STATUS_SIZE, *), IERROR	23
LUGIC	CAL FLAG		24
Retur	ns flag = true if all communic	cations associated with <i>active</i> handles in the array	25
	•	here no handle in the list is <i>active</i> ). In this case, each	26
-	(	<i>ive</i> request is set to the status of the corresponding	27

Returns flag = true if all communications associated with *active* handles in the array have *completed* (this includes the case where no handle in the list is *active*). In this case, each status entry that corresponds to an *active* request is set to the status of the corresponding operation. *Active persistent requests* are marked *inactive*. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI\_REQUEST\_NULL. Each status entry that corresponds to a *null* or *inactive* handle is set to *empty*.

Otherwise, flag = false is returned, no request is modified and the values of the status entries are undefined. This is a *local* procedure.

Errors that occurred during the execution of MPI\_TESTALL are handled in the same manner as errors in MPI\_WAITALL.

 $^{28}$ 

29

30

31

32

33

34

$\frac{1}{2}$	MPI_WAIT	SOME(incount, array_of_requ	ests, outcount, array_of_indices, array_of_statuses)
3	IN	incount	length of array_of_requests (non-negative integer)
4 5	INOUT	array_of_requests	array of requests (array of handles)
6	OUT	outcount	number of completed requests (integer)
7 8 9	OUT	array_of_indices	array of indices of operations that completed (array of integers)
9 10 11	OUT	array_of_statuses	array of status objects for operations that completed (array of status)
12		_	
13 14 15 16	C binding int MPI_W	-	•
17 18 19 20 21 22 23 24 25	MPI_Waits INTEG TYPE( INTEG TYPE(	array_of_statuses, i ER, INTENT(IN) :: incoun	t UT) :: array_of_requests(incount) unt, array_of_indices(*) statuses(*)
26 27 28 29 30		OME(INCOUNT, ARRAY_OF_RE ARRAY_OF_STATUSES, I ER INCOUNT, ARRAY_OF_REQ	QUESTS, OUTCOUNT, ARRAY_OF_INDICES, ERROR) UESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), PI_STATUS_SIZE, *), IERROR
<ol> <li>31</li> <li>32</li> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> <li>47</li> <li>48</li> </ol>	completed. have comp indices of t from zero i array_of_st requests ar and the ass If the = MPI_UNI When is desirable outcount, a all commu MPI_ERR_II success or	Returns in outcount the num leted. Returns in the first or these operations (index within n C and from one in Fortran). atuses the status for these c e marked as <i>inactive</i> . Any of sociated handle is set to MPI_ list contains no <i>active</i> handle DEFINED. one or more of the communic e to return specific information mray_of_indices and array_of_ nications that have succeeden N_STATUS and the error field to indicate the specific error	tions associated with <i>active</i> handles in the list have aber of requests from the list array_of_requests that atcount locations of the array array_of_indices the in the array array_of_requests; the array is indexed Returns in the first outcount locations of the array <i>ompleted</i> operations. <i>Completed active persistent</i> ther type or request that <i>completed</i> is deallocated, REQUEST_NULL. es, then the call returns <i>immediately</i> with outcount eations <i>completed</i> by MPI_WAITSOME fails, then it on on each communication. The arguments statuses will be adjusted to indicate <i>completion</i> of ed or failed. The call will return the error code d of each status returned will be set to indicate that occurred. The call will return MPI_SUCCESS will return another error code if it failed for other

reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

MPI\_TESTSOME(incount, array\_of\_requests, outcount, array\_of\_indices, array\_of\_statuses)

			6	
IN	incount	length of array_of_requests (non-negative integer)	7	
INOUT	array_of_requests	array of requests (array of handles)	8	
OUT	outcount	number of completed requests (integer)	9 10	
			11	
OUT	array_of_indices	array of indices of operations that completed (array of integers)	12	
			13	
OUT	array_of_statuses	array of status objects for operations that completed	14	
		(array of status)	15	
			16	
C binding			17	
int MPI_I		[_Request array_of_requests[],	18	
	int *outcount, int a	•	19	
	MPI_Status array_of	_statuses[])	20	
Fortran 2008 binding			21	
MPI_Tests	<pre>some(incount, array_of_re</pre>	equests, outcount, array_of_indices,	22	
array_or_statuses, lerror)			23	
	ER, INTENT(IN) :: incour		24	
	_	<pre>DUT) :: array_of_requests(incount)</pre>	25	
	-	<pre>ount, array_of_indices(*)</pre>	26	
	MPI_Status) :: array_of_		27 28	
INTEG	ER, OPTIONAL, INTENT(OUT	[) :: ierror	28 29	
Fortran b	oinding		30	
MPI_TESTS	SOME(INCOUNT, ARRAY_OF_RE	EQUESTS, OUTCOUNT, ARRAY_OF_INDICES,	31	
	ARRAY_OF_STATUSES,	IERROR)	32	
INTEG	ER INCOUNT, ARRAY_OF_REG	QUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),	33	
	ARRAY_OF_STATUSES(N	MPI_STATUS_SIZE, *), IERROR	34	
Behav	es like MPI_WAITSOMF_exc	cept that it returns <i>immediately</i> . If no operation has	35	
	,	here is no <i>active</i> handle in the list it returns <b>outcount</b>	36	
$=$ MPI_UN			37	
		lure, which returns <i>immediately</i> , whereas	38	
	-	mmunication <i>completes</i> , if it was passed a list that	39	
contains at least one <i>active</i> handle. Both calls fulfill a <b>fairness requirement</b> : If a request				

contains at least one *active* handle. Both calls fulfill a **fairness requirement**: If a request for a receive repeatedly appears in a list of requests passed to MPI\_WAITSOME or MPI\_TESTSOME, and a matching send has been posted, then the receive will eventually succeed, unless the send is satisfied by another receive; and similarly for send requests.

Errors that occur during the execution of MPI\_TESTSOME are handled as for MPI\_WAITSOME.

*Advice to users.* The use of MPI\_TESTSOME is likely to be more efficient than the use of MPI\_TESTANY. The former returns information on all *completed* communications,

1 2

 $\frac{3}{4}$ 

5

 $^{41}$ 

42

43

44

45

46

 $47 \\ 48$ 

with the latter, a new call is required for each communication that completes.
A server with multiple clients can use MPI\_WAITSOME so as not to starve any client.
Clients send messages to the server with service requests. The server calls
MPI\_WAITSOME with one receive request for each client, and then handles all receives that completed. If a call to MPI\_WAITANY is used instead, then one client could starve while requests from another client always sneak in first. (*End of advice to users.*)

Advice to implementors. MPI\_TESTSOME should complete as many pending communications as possible. (End of advice to implementors.)

```
Example 3.16 Client-server code (starvation can occur).
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .GT. 0) THEN
                               ! client code
   DO WHILE(.TRUE.)
      CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
      CALL MPI_WAIT(request, status, ierr)
   END DO
ELSE
             ! rank=0 -- server code
   DO i=1, size-1
      CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
                     comm, request_list(i), ierr)
   END DO
   DO WHILE(.TRUE.)
      CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
      CALL DO_SERVICE(a(1,index)) ! handle one message
      CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag, &
                     comm, request_list(index), ierr)
   END DO
END IF
```

```
Example 3.17 Same code, using MPI_WAITSOME.
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .GT. 0) THEN
                               ! client code
   DO WHILE(.TRUE.)
      CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
      CALL MPI_WAIT(request, status, ierr)
   END DO
ELSE
             ! rank=0 -- server code
   DO i=1, size-1
      CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
                     comm, request_list(i), ierr)
   END DO
   DO WHILE(.TRUE.)
      CALL MPI_WAITSOME(size, request_list, numdone, &
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9 10

11 12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31 32

33 34

35

36

37

38

39

40

41

42

43

44

45

46

47

## 3.7.6 Non-Destructive Test of status

This call is useful for accessing the information associated with a request, without *freeing* the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same *completed* request and extract from it the status information.

MPI\_REQUEST\_GET\_STATUS(request, flag, status)

IN	request	request (handle)
OUT	flag	boolean flag, same as from $MPI\_TEST$ (logical)
OUT	status	status object if flag is true (status)

#### C binding

#### Fortran 2008 binding

MPI\_Request\_get\_status(request, flag, status, ierror)
 TYPE(MPI\_Request), INTENT(IN) :: request
 LOGICAL, INTENT(OUT) :: flag
 TYPE(MPI\_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# Fortran binding

```
MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
LOGICAL FLAG
```

Sets flag = true if the operation is *complete*, and, if so, returns in status the request status. However, unlike test or wait, it does not deallocate or *inactivate* the request; a subsequent call to test, wait or free should be executed with that request. It sets flag = false if the operation is not *complete*.

One is allowed to call MPI\_REQUEST\_GET\_STATUS with a *null* or *inactive* request argument. In such a case the procedure returns with flag = true and *empty* status.

 $\mathbf{2}$ 

3.8 Pro	obe and Cancel		
The MPI_PROBE, MPI_IPROBE, MPI_MPROBE, and MPI_IMPROBE procedures allow in- coming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the <b>probe</b> (basically, the information returned by <b>status</b> ). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message. The MPI_CANCEL procedure allows pending communications to be <b>cancelled</b> . This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a <i>cancel</i> may be needed to free these resources gracefully. <i>Cancelling</i> a send request by calling MPI_CANCEL is deprecated. <i>Cancelling</i> a send- recv request by calling MPI_CANCEL is not allowed.			
3.8.1 Pro	obe		
MPI_IPRO	BE(source, tag, comm, flag, st	atus)	
IN	source	rank of source or $MPI\_ANY\_SOURCE$ (integer)	
IN	tag	message tag or $MPI\_ANY\_TAG$ (integer)	
IN	comm	communicator (handle)	
OUT	flag	true if there is a matching message that can be received (logical)	
OUT	status	status object (status)	
	0	g, MPI_Comm comm, int *flag,	
Fortran 2	2008 binding		
-	-	-	
		•	
		Comm	
INTEC	GER, OPTIONAL, INTENT(OUT)	) :: ierror	
	0		
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR LOGICAL FLAG MPI_IPROBE returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV with the same argument values for source, tag, comm, and status executed at the same point in the program, and returns in status the same value that would have been returned by MPI_RECV. Otherwise, the call returns flag = false, and lagues status undefined			
	The MPI_ coming madecide how the inform receive but The M is required buffers), a <i>Cance</i> recv reque 3.8.1 Pre 3.8.1 Pre 3.8.1 Pre MPI_IPRC IN IN IN OUT OUT C bindin int MPI_1 Fortran 2 MPI_IPRO INTEC TYPE LOGIC TYPE LOGIC TYPE LOGIC TYPE LOGIC TYPE LOGIC	The MPI_PROBE, MPI_IPROBE, MPI_M coming messages to be checked for, with decide how to receive them, based on the information returned by status). In receive buffer, according to the length of The MPI_CANCEL procedure allow is required for cleanup. Posting a send buffers), and a <i>cancel</i> may be needed to <i>Cancelling</i> a send request by callin recv request by calling MPI_CANCEL is 3.8.1 Probe MPI_IPROBE(source, tag, comm, flag, st IN source IN tag IN comm OUT flag OUT status <b>C binding</b> int MPI_Iprobe(int source, int tag MPI_Status *status) Fortran 2008 binding MPI_Iprobe(source, tag, comm, flag INTEGER, INTENT(IN) :: source TYPE(MPI_Comm), INTENT(IN) :: LOGICAL, INTENT(OUT) :: flag TYPE(MPI_Status) :: status INTEGER, OPTIONAL, INTENT(OUT) Fortran binding MPI_IPROBE(SOURCE, TAG, COMM, STA LOGICAL FLAG MPI_IPROBE returns flag = true if matches the pattern specified by the argu- same message that would have been recei- values for source, tag, comm, and status	

If MPI\_IPROBE returns flag = true, then the content of the status object can be subsequently accessed as described in Section 3.2.5 to find the source, tag, and length of the probed message.

MPI\_IPROBE is a *local* procedure since its return does not depend on MPI calls in other MPI processes, which is marked with the prefix I (for *immediate*).

A subsequent receive executed with the same communicator, and the source and tag returned in status by MPI\_IPROBE will receive the message that was matched by the probe, if no other intervening receive occurs after the probe, and the send is not successfully *cancelled* before the receive. If the receiving process is multithreaded, it is the user's responsibility to ensure that the last condition holds.

The source argument of MPI\_IPROBE can be MPI\_ANY\_SOURCE, and the tag argument can be MPI\_ANY\_TAG, so that one can *probe* for *messages* from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.

A probe with MPI\_PROC\_NULL as source returns flag = true, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0; see Section 3.10.

MPI\_PROBE(source, tag, comm, status)

IN	source	rank of source or $MPI_ANY_SOURCE$ (integer)
IN	tag	message tag or $MPI\_ANY\_TAG$ (integer)
IN	comm	communicator (handle)
OUT	status	status object (status)

#### C binding

int MPI\_Probe(int source, int tag, MPI\_Comm comm, MPI\_Status \*status)

#### Fortran 2008 binding

MPI\_Probe(source, tag, comm, status, ierror)
 INTEGER, INTENT(IN) :: source, tag
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 TYPE(MPI\_Status) :: status
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

```
MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
```

INTEGER SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR

MPI\_PROBE behaves like MPI\_IPROBE except that it is a *non-local* call that returns only after a matching message has been found.

The MPI implementation of MPI\_PROBE and MPI\_IPROBE needs to guarantee *progress*: 43 if a call to MPI\_PROBE has been issued by a process, and a send that matches the probe 44 has been *initiated* by some process, then the call to MPI\_PROBE will return, unless the 45 message is received by another concurrent receive operation (that is executed by another 46 thread at the probing process). 47

1

2

3

4

5

6 7

8 9

10

11

12

13 14

15

16

17

18 19 20

27 28

29

30

31

32

33

34

35

36 37

38

39

40

41

42

Similarly, if a process busy waits with MPI\_IPROBE and a matching message has been issued, then the call to MPI\_IPROBE will eventually return flag = true unless the message is received by another concurrent receive operation or matched by a concurrent *matching* probe.

```
Example 3.18 Use probe to wait for an incoming message.
    CALL MPI_COMM_RANK(comm, rank, ierr)
    IF (rank .EQ. 0) THEN
       CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
    ELSE IF (rank .EQ. 1) THEN
       CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
    ELSE IF (rank .EQ. 2) THEN
       DO i=1,2
          CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
                          comm, status, ierr)
          IF (status(MPI_SOURCE) .EQ. 0) THEN
100
             CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
          ELSE
200
             CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
          END IF
       END DO
    END IF
```

Each message is received with the right type.

```
Example 3.19 A similar program to the previous example, but now it has a problem.
! ----- THIS EXAMPLE IS ERRONEOUS ------
    CALL MPI_COMM_RANK(comm, rank, ierr)
    IF (rank .EQ. 0) THEN
       CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
    ELSE IF (rank .EQ. 1) THEN
       CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
    ELSE IF (rank .EQ. 2) THEN
       DO i=1,2
          CALL MPI_PROBE(MPI_ANY_SOURCE, 0, &
                         comm, status, ierr)
          IF (status(MPI_SOURCE) .EQ. 0) THEN
100
             CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, &
                           0, comm, status, ierr)
          ELSE
200
             CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, &
                           0, comm, status, ierr)
          END IF
       END DO
    END IF
```

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23 24

 $25 \\ 26$ 

27 28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

In Example 3.19, the two receive calls in statements labeled 100 and 200 in Example 3.18 are slightly modified, using MPI\_ANY\_SOURCE as the source argument. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to MPI\_PROBE.

Advice to users. In a multithreaded MPI program, MPI\_PROBE and MPI\_IPROBE might need special care. If a thread *probes* for a message and then immediately posts a matching receive, the receive may match a message other than that found by the probe since another thread could concurrently receive that original message [33]. MPI\_MPROBE and MPI\_IMPROBE solve this problem by matching the incoming message so that it may only be received with MPI\_MRECV or MPI\_IMRECV on the corresponding *message handle*. (*End of advice to users*.)

Advice to implementors. A call to MPI\_PROBE will match the message that would have been received by a call to MPI\_RECV with the same argmument values for source, tag, comm, and status executed at the same point. Suppose that this message has source s, tag t and communicator c. If the tag argument in the probe call has value MPI\_ANY\_TAG then the message probed will be the earliest pending message from source s with communicator c and any tag; in any case, the message probed will be the earliest pending message from source s with tag t and communicator c (this is the message that would have been received, so as to preserve message order). This message continues as the earliest pending message from source s with tag t and communicator c, until it is received. A receive operation subsequent to the probe that uses the same communicator as the probe and uses the tag and source values returned by the probe, must receive this message, unless it has already been received by another receive operation. (*End of advice to implementors*.)

#### 3.8.2 Matching Probe

The function MPI\_PROBE checks for incoming *messages* without receiving them. Since the list of incoming *messages* is global among the threads of each MPI process, it can be hard to use this functionality in threaded environments [33, 30].

Like MPI\_PROBE and MPI\_IPROBE, the **matching probe** operation (MPI\_MPROBE and MPI\_IMPROBE procedures) allow incoming *messages* to be queried without actually receiving them, except that MPI\_MPROBE and MPI\_IMPROBE provide a mechanism to receive the specific *message* that was matched regardless of other intervening probe or receive operations. This gives the application an opportunity to decide how to receive the message, based on the information returned by the probe. In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13 14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27 28

29

30

31

32

33

34

35

36

37

38

1 MPI\_IMPROBE(source, tag, comm, flag, message, status) 2 IN source rank of source or MPI\_ANY\_SOURCE (integer) 3 IN message tag or MPI\_ANY\_TAG (integer) tag 4 5IN communicator (handle) comm 6 OUT true if there is a matching message that can be flag 7 received (logical) 8 OUT message returned message (handle) 9 10 OUT status status object (status) 11 12C binding 13 int MPI\_Improbe(int source, int tag, MPI\_Comm comm, int \*flag, 14MPI\_Message \*message, MPI\_Status \*status) 15Fortran 2008 binding 16MPI\_Improbe(source, tag, comm, flag, message, status, ierror) 17INTEGER, INTENT(IN) :: source, tag 18 TYPE(MPI\_Comm), INTENT(IN) :: comm 19LOGICAL, INTENT(OUT) :: flag 20TYPE(MPI\_Message), INTENT(OUT) :: message 21TYPE(MPI\_Status) :: status 22 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 23 $^{24}$ Fortran binding 25MPI\_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR) 26INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 27LOGICAL FLAG 28MPI\_IMPROBE returns flag = true if there is a message that can be received and that 29 matches the pattern specified by the arguments source, tag, and comm. The call matches the 30 same message that would have been received by a call to MPI\_RECV with the same argument  $^{31}$ values for source, tag, comm, and status executed at the same point in the program and 32 returns in status the same value that would have been returned by MPI\_RECV. In addition, 33 34it returns in message a message handle to the matched message. Otherwise, the call returns flag = false, and leaves status and message undefined. 35 MPI\_IMPROBE is a *local* procedure. According to the definitions in Section 2.4.2 and 36 in contrast to MPI\_IPROBE, it is a *nonblocking* procedure because it is the *initialization* of 37 a matched receive operation. 38 A matched receive (MPI\_MRECV or MPI\_IMRECV) executed with the message handle 39 will receive the message that was matched by the *matching probe*. Unlike MPI\_IPROBE, no 40other probe or receive operation may match the message returned by MPI\_IMPROBE. Each 41 message handle returned by MPI\_IMPROBE must be received with either MPI\_MRECV or 42MPI\_IMRECV. 43 The source argument of MPI\_IMPROBE can be MPI\_ANY\_SOURCE, and the 44tag argument can be MPI\_ANY\_TAG, so that one can probe for messages from an arbitrary 45

source and/or with an arbitrary tag. However, a specific communication context must be
 provided with the comm argument.

88

A synchronous mode send operation that is matched with MPI_IMPROBE or MPI_MPROBE will complete successfully only if both a matching receive is posted with MPI_MRECV or MPI_IMRECV, and the matching receive operation has started to receive the message sent by the synchronous mode send. There is a special <b>predefined message handle</b> : MPI_MESSAGE_NO_PROC, which is a message which has MPI_PROC_NULL as its source process. The predefined constant MPI_MESSAGE_NULL is the value used for <b>invalid message handles</b> . A matching probe with source = MPI_PROC_NULL returns flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 3.10. It is not necessary to call MPI_MRECV or MPI_IMRECV with MPI_MESSAGE_NO_PROC, but it is not erroneous to do so. Rationale. MPI_MESSAGE_NO_PROC, but it is not erroneous to do so.			
			17 18
	OBE(source, tag, comm, messa	<b>.</b> ,	19
IN	source	rank of source or MPI_ANY_SOURCE (integer)	20 21
IN	tag	message tag or $MPI_ANY_TAG$ (integer)	21 22
IN	comm	communicator (handle)	23
OUT	message	returned message (handle)	24
OUT	status	status object (status)	25
			26
C binding int MPI_M		g, MPI_Comm comm, MPI_Message *message,	27 28 29 30
Fortran 2	2008 binding		31
	pe(source, tag, comm, mes	sage, status, ierror)	32
-	ER, INTENT(IN) :: source	-	33
TYPE(	MPI_Comm), INTENT(IN) ::	comm	34
	MPI_Message), INTENT(OUT	) :: message	35
	MPI_Status) :: status ER, OPTIONAL, INTENT(OUT		36
INTEG	ER, UPIIUNAL, INIENI(UUI	) :: lerror	37 38
Fortran b	8		39
	BE(SOURCE, TAG, COMM, MES EER SOURCE, TAG, COMM, ME	SAGE, STATUS, IERROR) SSAGE, STATUS(MPI_STATUS_SIZE), IERROR	40
only after The ir in the sam	a matching message has been nplementation of MPI_MPRO e way as in the case of MPI_F ding to the definitions in Sect	<b>BE</b> and <b>MPI_IMPROBE</b> needs to guarantee <i>progress</i>	41 42 43 44 45 46 47 48

```
1
                            This is one of the exceptions in which incomplete procedures are
           Advice to users.
\mathbf{2}
           non-local. (End of advice to users.)
3
4
     3.8.3
            Matched Receives
5
     The matched receive operation (MPI_MRECV and MPI_IMRECV procedures) receive mes-
6
     sages that have been previously matched by a matching probe operation (Section 3.8.2).
7
8
9
     MPI_MRECV(buf, count, datatype, message, status)
10
       OUT
                 buf
                                             initial address of receive buffer (choice)
11
12
       IN
                 count
                                             number of elements in receive buffer (non-negative
13
                                             integer)
14
       IN
                                             datatype of each receive buffer element (handle)
                 datatype
15
       INOUT
                 message
                                             message (handle)
16
17
       OUT
                                             status object (status)
                 status
18
19
     C binding
20
     int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype,
21
                    MPI_Message *message, MPI_Status *status)
22
     int MPI_Mrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,
23
                    MPI_Message *message, MPI_Status *status)
^{24}
25
     Fortran 2008 binding
26
     MPI_Mrecv(buf, count, datatype, message, status, ierror)
27
          TYPE(*), DIMENSION(..) :: buf
28
          INTEGER, INTENT(IN) :: count
29
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
          TYPE(MPI_Message), INTENT(INOUT) :: message
31
          TYPE(MPI_Status) :: status
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Mrecv(buf, count, datatype, message, status, ierror) !(_c)
          TYPE(*), DIMENSION(..) :: buf
35
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
36
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
          TYPE(MPI_Message), INTENT(INOUT) :: message
38
          TYPE(MPI_Status) :: status
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     Fortran binding
42
     MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
43
          <type> BUF(*)
44
          INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
45
          This call receives a message matched by a matching probe operation (Section 3.8.2).
46
47
          The receive buffer consists of the storage containing count consecutive elements of the
     type specified by datatype, starting at address buf. The length of the received message must
48
```

be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

If the message is shorter than the receive buffer, then only those locations corresponding to the (shorter) message are modified.

On return from this function, the *message handle* is set to MPI\_MESSAGE\_NULL. All errors that occur during the execution of this operation are handled according to the error handler set for the communicator used in the matching probe call that produced the message handle.

If MPI\_MRECV is called with MPI\_MESSAGE\_NO\_PROC as the message argument, the call returns immediately with the status object set to source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0. This is consistent with the status object produced by a call to MPI\_RECV or to MPI\_PROBE with source = MPI\_PROC\_NULL (see Section 3.10). A call to MPI\_MRECV with MPI\_MESSAGE\_NULL is *erroneous*.

MPI\_IMRECV(buf, count, datatype, message, request)

OUT	buf	initial address of receive buffer (choice)	17
IN	count	number of elements in receive buffer (non-negative integer)	18 19 20
IN	datatype	datatype of each receive buffer element (handle)	20 21
INOUT	message	message (handle)	22
OUT	request	communication request (handle)	23 24

#### C binding

<pre>int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,</pre>	27
MPI_Message *message, MPI_Request *request)	28
<pre>int MPI_Imrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	29
MPI_Message *message, MPI_Request *request)	30
	31
Fortran 2008 binding	32
MPI_Imrecv(buf, count, datatype, message, request, ierror)	33
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	34
INTEGER, INTENT(IN) :: count	35
TYPE(MPI_Datatype), INTENT(IN) :: datatype	36
TYPE(MPI_Message), INTENT(INOUT) :: message	37
TYPE(MPI_Request), INTENT(OUT) :: request	38
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	39
<pre>MPI_Imrecv(buf, count, datatype, message, request, ierror) !(_c)</pre>	40
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	41
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	42
TYPE(MPI_Datatype), INTENT(IN) :: datatype	43
TYPE(MPI_Message), INTENT(INOUT) :: message	44
TYPE(MPI_Request), INTENT(OUT) :: request	45
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	46
,,,,,,	47
	48

1	Fortran binding			
2	5			
3	MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)			
	<type> BUF(*)</type>			
4	INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR			
5	MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking			
6	receive of a matched message. Completion semantics are similar to MPI_IRECV as described			
7				
8	in Section 3.7.2. On return from this function, the <i>message handle</i> is set to MPI_MESSAGE_NULL.			
9				
10	If MPI_IMRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the			
11	call returns immediately with a request object which, when completed, will yield a status			
12	object set to source = $MPI_PROC_NULL$ , tag = $MPI_ANY_TAG$ , and count = 0, as if a receive			
13	from MPI_PROC_NULL was issued (see Section $3.10$ ). A call to MPI_IMRECV with			
14	MPI_MESSAGE_NULL is <i>erroneous</i> .			
15				
16	Advice to implementors. If reception of a matched message is started with			
17	MPI_IMRECV, then it is possible to <i>cancel</i> the returned request with MPI_CANCEL. If			
18	MPI_CANCEL succeeds, the matched message must be found by a subsequent message			
19	probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by			
	a subsequent receive operation or <i>cancelled</i> by the sender. See Section 3.8.4 for details			
20	about MPI_CANCEL. The <i>cancellation</i> of operations initiated with MPI_IMRECV may			
21	fail. (End of advice to implementors.)			
22				
23	3.8.4 Cancel			
24				
25				
26				
27	MPI_CANCEL(request)			
28	IN request communication request (handle)			
29				
30	C binding			
31				
32	<pre>int MPI_Cancel(MPI_Request *request)</pre>			
33	Fortran 2008 binding			
34	MPI_Cancel(request, ierror)			
35	TYPE(MPI_Request), INTENT(IN) :: request			
36	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
37				
38	Fortran binding			
39	MPI_CANCEL(REQUEST, IERROR)			
40	INTEGER REQUEST, IERROR			
40	A call to MDI CANCEL maybe for concellation a new line way blocking of			
	A call to MPI_CANCEL marks for <i>cancellation</i> a pending, <i>nonblocking</i> communica-			
42	tion operation (send or receive). <i>Cancelling</i> a send request by calling MPI_CANCEL is			
43	deprecated. The <i>cancel</i> call is <i>local</i> . It returns <i>immediately</i> , possibly before the communi-			
44	cation is actually <i>cancelled</i> . It is still necessary to call MPI_REQUEST_FREE, MPI_WAIT or			
45	MPI_TEST (or any of the derived procedures) with the <i>cancelled</i> request as argument after			
46	the call to MPI_CANCEL. If a communication is marked for <i>cancellation</i> , then a MPI_WAIT call for that communication is guaranteed to return imagnetive of the activities of other			
47				

call for that communication is guaranteed to return, irrespective of the activities of other
 processes (i.e., MPI\_WAIT behaves as a *local* function); similarly if MPI\_TEST is repeatedly

called in a busy wait loop for a *cancelled* communication, then MPI\_TEST will eventually be successful.

MPI\_CANCEL can be used to *cancel* a communication that uses a *persistent commu*nication request (see Section 3.9), in the same way it is used for nonpersistent requests. *Cancelling* a persistent send request by calling MPI\_CANCEL is deprecated. A successful *cancellation cancels* the *active* communication, but not the request itself. After the call to MPI\_CANCEL and the subsequent call to MPI\_WAIT or MPI\_TEST, the request becomes *inactive* and can be activated for a new communication.

The successful *cancellation* of a *buffered mode send* frees the buffer space occupied by the pending message. *Cancelling* a *buffered mode send* request by calling MPI\_CANCEL is deprecated.

Either the *cancellation* succeeds, or the communication succeeds, but not both. If a send is marked for *cancellation*, which is deprecated, then it must be the case that either the send *completes* normally, in which case the message sent was received at the destination process, or that the send is successfully *cancelled*, in which case no part of the message was received at the destination. Then, any matching receive has to be satisfied by another send. If a receive is marked for *cancellation*, then it must be the case that either the receive *completes* normally, or that the receive is successfully *cancelled*, in which case no part of the receive buffer is altered. Then, any matching send has to be satisfied by another receive.

If the operation has been *cancelled*, then information to that effect will be returned in the status argument of the operation that *completes* the communication.

Rationale. Although the IN request handle parameter should not need to be passed by reference, the C binding has listed the argument type as MPI\_Request\* since MPI-1.0. This function signature therefore cannot be changed without breaking existing MPI applications. (*End of rationale.*)

IN	status	status object (status)
OUT	flag	true if the operation has been cancelled (logical)

#### C binding

```
int MPI_Test_cancelled(const MPI_Status *status, int *flag)
```

#### Fortran 2008 binding

```
MPI_Test_cancelled(status, flag, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

# Fortran binding

```
MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), IERROR
LOGICAL FLAG
```

Returns flag = true if the communication associated with the status object was *cancelled* <sup>47</sup> successfully. In such a case, all other fields of status (such as count or tag) are undefined. <sup>48</sup>

 $\mathbf{2}$ 

 $^{24}$ 

Returns flag = false, otherwise. If a receive operation might be *cancelled* then one should call MPI\_TEST\_CANCELLED first, to check whether the operation was *cancelled*, before checking on the other fields of the return status.

Advice to users. Cancel can be an expensive operation that should be used only exceptionally. (End of advice to users.)

Advice to implementors. If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the *cancellation* of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement

MPI\_CANCEL, this is still a *local* procedure, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (*End of advice to implementors.*)

3.9 Persistent Communication Requests

20Often a communication with the same argument list (with the exception of the buffer con-21tents) is repeatedly executed within the inner loop of a parallel computation. In such a 22situation, it may be possible to optimize the communication by binding the list of commu-23nication arguments to a *persistent communication request* once and then repeatedly using  $^{24}$ the request to start and complete operations. In the case of point-to-point communication, 25the *persistent communication request* thus created can be thought of as a communication 26port or a "half-channel." It does not provide the full functionality of a conventional channel, 27since there is no binding of the send port to the receive port. This construct allows reduction 28of the overhead for communication between the process and communication controller, but 29not of the overhead for communication between one communication controller and another. 30 It is not necessary that messages sent with a persistent point-to-point request be received  $^{31}$ by a receive operation using a persistent point-to-point request, or vice versa.

There are also persistent collective communication operations defined in Section 6.13 and Section 8.8. The remainder of this section covers the point-to-point persistent *initialization* operations and the start routines, which are used for persistent point-to-point, partitioned point-to-point, and persistent collective communication operations.

<sup>36</sup> A point-to-point **persistent communication request** is created using one of the five
 <sup>37</sup> following calls. These point-to-point persistent *initialization* calls involve no communica <sup>38</sup> tion.

- 39
- 40 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48

4 5 6

7

8

9

10

11

12

13

14

15

16 17 18

19

1

 $\mathbf{2}$ 

MPI_SEN	D_INIT(buf, count, datatype, de	est, tag, comm, request)	1
IN	buf	initial address of send buffer (choice)	2 3
IN	count	number of elements sent (non-negative integer)	4
IN	datatype	type of each element (handle)	5
IN	dest	rank of destination (integer)	6
IN	tag	message tag (integer)	7 8
IN	comm	communicator (handle)	9
OUT	request	communication request (handle)	10 11
	Send_init(const void *buf,	int count, MPI_Datatype datatype, PI_Comm comm, MPI_Request *request) of MPI_Count count	12 13 14 15 16
1110 111 1_1		e, int dest, int tag, MPI_Comm comm,	17 18
MPI_Send_ TYPE( INTEC TYPE( TYPE)		:: datatype comm :: request	20 21 22 23 24 25 26 27
<pre>MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)    TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count    TYPE(MPI_Datatype), INTENT(IN) :: datatype    INTEGER, INTENT(IN) :: dest, tag    TYPE(MPI_Comm), INTENT(IN) :: comm    TYPE(MPI_Request), INTENT(OUT) :: request    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			28 29 30 31 32 33 34 35
<type INTEC</type 	INIT(BUF, COUNT, DATATYPE >> BUF(*) ER COUNT, DATATYPE, DEST,	E, DEST, TAG, COMM, REQUEST, IERROR) TAG, COMM, REQUEST, IERROR request for a standard mode send operation.	36 37 38 39 40 41 42 43
			44

1 MPI\_BSEND\_INIT(buf, count, datatype, dest, tag, comm, request)  $\mathbf{2}$ IN buf initial address of send buffer (choice) 3 IN count number of elements sent (non-negative integer) 4 5IN type of each element (handle) datatype 6 IN dest rank of destination (integer) 7 IN message tag (integer) tag 8 9 IN communicator (handle) comm 10 OUT request communication request (handle) 11 12C binding 13 int MPI\_Bsend\_init(const void \*buf, int count, MPI\_Datatype datatype, 14int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 1516int MPI\_Bsend\_init\_c(const void \*buf, MPI\_Count count, 17 MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm, 18 MPI\_Request \*request) 19Fortran 2008 binding 20MPI\_Bsend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 21TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf 22 INTEGER, INTENT(IN) :: count, dest, tag 23TYPE(MPI\_Datatype), INTENT(IN) :: datatype 24TYPE(MPI\_Comm), INTENT(IN) :: comm 25TYPE(MPI\_Request), INTENT(OUT) :: request 26INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2728MPI\_Bsend\_init(buf, count, datatype, dest, tag, comm, request, ierror) 29!(\_c) 30 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf 31INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 32 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 33 INTEGER, INTENT(IN) :: dest, tag 34 TYPE(MPI\_Comm), INTENT(IN) :: comm 35TYPE(MPI\_Request), INTENT(OUT) :: request 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 Fortran binding 38 MPI\_BSEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 39 <type> BUF(\*) 40INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 41 42Creates a persistent communication request for a buffered mode send operation. 43 444546 47 48

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)			
IN	buf	initial address of send buffer (choice)	2 3
IN	count	number of elements sent (non-negative integer)	4
IN	datatype	type of each element (handle)	5
IN	dest	rank of destination (integer)	6
IN	tag	message tag (integer)	7 8
IN	comm	communicator (handle)	9
			10
OUT	request	communication request (handle)	11
C bind	ing		12
	0	<pre>void *buf, int count, MPI_Datatype datatype,</pre>	13 14
	int dest, i	nt tag, MPI_Comm comm, MPI_Request *request)	15
int MP	[_Ssend_init_c(cons	st void *buf, MPI_Count count,	16
	• •	e datatype, int dest, int tag, MPI_Comm comm,	17
	MPI_Request	*request)	18 19
Fortra	n 2008 binding		20
		t, datatype, dest, tag, comm, request, ierror)	21
		.), INTENT(IN), ASYNCHRONOUS :: buf	22
INTEGER, INTENT(IN) :: count, dest, tag TYPE(MPI_Datatype), INTENT(IN) :: datatype			23
TYPE(MPI_Comm), INTENT(IN) :: comm			24 25
TYPE(MPI_Request), INTENT(OUT) :: request			26
IN	FEGER, OPTIONAL, IN	NTENT(OUT) :: ierror	27
MPI_Sse	end_init(buf, count	t, datatype, dest, tag, comm, request, ierror)	28
	!(_c)		29
		.), INTENT(IN), ASYNCHRONOUS :: buf	30 31
		NT_KIND), INTENT(IN) :: count INTENT(IN) :: datatype	32
	reger, INTENT(IN)		33
	PE(MPI_Comm), INTEN	, 0	34
TYI	PE(MPI_Request), IN	NTENT(OUT) :: request	35
INT	FEGER, OPTIONAL, IN	NTENT(OUT) :: ierror	36 37
Fortra	n binding		38
MPI_SSI	END_INIT(BUF, COUNT	Γ, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)	39
•	<type> BUF(*)</type>		
1N.	TEGER COUNT, DATATY	YPE, DEST, TAG, COMM, REQUEST, IERROR	41
Cre	eates a <i>persistent com</i>	munication request for a synchronous mode send operation.	42 43
			43 44
			45

```
1
     MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)
\mathbf{2}
       IN
                buf
                                            initial address of send buffer (choice)
3
       IN
                count
                                            number of elements sent (non-negative integer)
4
5
       IN
                                            type of each element (handle)
                datatype
6
       IN
                dest
                                            rank of destination (integer)
7
       IN
                                            message tag (integer)
                tag
8
9
       IN
                                            communicator (handle)
                comm
10
       OUT
                request
                                            communication request (handle)
11
12
     C binding
13
     int MPI_Rsend_init(const void *buf, int count, MPI_Datatype datatype,
14
                    int dest, int tag, MPI_Comm comm, MPI_Request *request)
15
16
     int MPI_Rsend_init_c(const void *buf, MPI_Count count,
17
                    MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
18
                    MPI_Request *request)
19
     Fortran 2008 binding
20
     MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
21
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
22
         INTEGER, INTENT(IN) :: count, dest, tag
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
29
                    !(_c)
30
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
31
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         INTEGER, INTENT(IN) :: dest, tag
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     Fortran binding
38
     MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
39
         <type> BUF(*)
40
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
41
42
         Creates a persistent communication request for a ready mode send operation.
43
44
45
46
47
48
```

MPI_REC	√_INIT(buf, count, datatype, so	ource, tag, comm, request)	1
OUT	buf	initial address of receive buffer (choice)	2 3
IN	count	number of elements received (non-negative integer)	4
IN	datatype	type of each element (handle)	5
IN	source	rank of source or MPI_ANY_SOURCE (integer)	6
IN	tag	message tag or MPI_ANY_TAG (integer)	7 8
IN	comm	communicator (handle)	9
OUT	request	communication request (handle)	10
		• • • •	11 12
C bindin	•		13
int MPI_		count, MPI_Datatype datatype, int source, mm, MPI_Request *request)	14 15
int MPI_		I_Count count, MPI_Datatype datatype, MPI_Comm comm, MPI_Request *request)	16 17
Fortron	2008 binding		18
	6	e, source, tag, comm, request, ierror)	19 20
TYPE	(*), DIMENSION(), ASYNC	HRONOUS :: buf	21
	GER, INTENT(IN) :: count,	•	22
	(MPI_Datatype), INTENT(IN) (MPI_Comm), INTENT(IN) ::		23
	(MPI_Request), INTENT(OUT)		24 25
	GER, OPTIONAL, INTENT(OUT)	•	26
MPI_Recv	_init(buf, count, datatype	e, source, tag, comm, request, ierror)	27
	!(_c)		28 29
	(*), DIMENSION(), ASYNCI		30
	GER(KIND=MPI_COUNT_KIND), (MPI_Datatype), INTENT(IN)		31
	GER, INTENT(IN) :: source		32
TYPE	(MPI_Comm), INTENT(IN) ::	comm	33
	(MPI_Request), INTENT(OUT)	-	34 35
TNLE(	GER, OPTIONAL, INTENT(OUT)	) :: lerror	36
Fortran	0		37
		E, SOURCE, TAG, COMM, REQUEST, IERROR)	38
• 1	e> BUF(*) GER COUNT. DATATYPE. SOUR(	CE, TAG, COMM, REQUEST, IERROR	39 40
			40
	-	request for a receive operation. The argument buf	42
	is marked as OUT because the user gives permission to write on the receive buffer by passing the argument to MPI_RECV_INIT.		
-		is <i>inactive</i> after it was created—no active commu-	44
	attached to the request.		45 46
A cor tion MPI_	_	tent communication request is started by the func-	40

```
1
     MPI_START(request)
2
       INOUT
                 request
                                              communication request (handle)
3
4
     C binding
5
     int MPI_Start(MPI_Request *request)
6
\overline{7}
     Fortran 2008 binding
8
     MPI_Start(request, ierror)
9
          TYPE(MPI_Request), INTENT(INOUT) :: request
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     Fortran binding
12
     MPI_START(REQUEST, IERROR)
13
          INTEGER REQUEST, IERROR
14
15
          The argument, request, is a handle returned by any of the initialization procedures for
16
     persistent point-to-point communication (the previous five procedures), or for partitioned
17
     point-to-point communication (see Section 4), or for persistent collective communication
18
     (see Sections 6.13 and 8.8). The associated request should be inactive. The request becomes
19
     active once the call is made.
20
          If the request is for a ready mode send operation, then a matching receive operation
21
     should be posted before the call is made. The communication buffer should not be modified
22
     after the call, and until the operation completes.
23
          The call is local, with similar semantics to the nonblocking communication operations
^{24}
     described in Section 3.7. That is, a call to MPI_START with a request created by
25
     MPI_SEND_INIT starts a communication in the same manner as a call to MPI_ISEND; a
26
     call to MPI_START with a request created by MPI_BSEND_INIT starts a communication
27
     in the same manner as a call to MPI_IBSEND; and so on.
28
29
     MPI_STARTALL(count, array_of_requests)
30
31
       IN
                                              list length (non-negative integer)
                 count
32
       INOUT
                 array_of_requests
                                              array of requests (array of handles)
33
34
     C binding
35
     int MPI_Startall(int count, MPI_Request array_of_requests[])
36
37
     Fortran 2008 binding
38
     MPI_Startall(count, array_of_requests, ierror)
39
          INTEGER, INTENT(IN) :: count
40
          TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
41
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     Fortran binding
43
     MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
44
          INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
45
46
          The execution of MPI_STARTALL has the same effect as the execution of MPI_START
47
     for each of the array elements in some arbitrary order. MPI_STARTALL with an array of
48
```

length one is equivalent to MPI\_START.

A communication started with a call to MPI\_START or MPI\_STARTALL is completed by a call to MPI\_WAIT, MPI\_TEST, or one of the derived functions described in Section 3.7.5. The request becomes *inactive* after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI\_START or MPI\_STARTALL call.

A persistent communication request is deallocated by a call to MPI\_REQUEST\_FREE (Section 3.7.3). The call to MPI\_REQUEST\_FREE can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes *inactive*. Active receive requests should not be *freed*. Otherwise, it will not be possible to check that the receive has *completed*. Collective operation requests (defined in Section 6.12 and Section 8.7 for nonblocking collective operations, and Section 6.13 and Section 8.8 for persistent collective operations) must not be *freed* while active. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form,

#### Create (Start Complete)\* Free

where \* indicates zero or more repetitions. If the same *persistent communication request* is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is obeyed.

A send operation *started* with MPI\_START can be matched with any receive operation and, likewise, a receive operation *started* with MPI\_START can receive messages generated by any send operation.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. (End of advice to users.)

# 3.10 Null Processes

In many instances, it is convenient to specify a "dummy" source or destination for communication. This simplifies the code that is needed for dealing with boundaries, for example, in the case of a noncircular shift done with calls to send-receive.

The special value MPI\_PROC\_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI\_PROC\_NULL has no effect. A send to MPI\_PROC\_NULL succeeds and returns as soon as possible. A receive from MPI\_PROC\_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI\_PROC\_NULL is executed then the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG and count = 0. A probe or matching probe with source = MPI\_PROC\_NULL succeeds and returns as soon as possible, and the status object returns source = MPI\_PROC\_NULL succeeds and returns as soon as possible, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG and count = 0. A matching probe (cf. Section 3.8.2) with source = MPI\_PROC\_NULL returns flag = true, message = MPI\_MESSAGE\_NO\_PROC, and the status object returns source = MPI\_PROC\_NULL, tag = MPI\_PROC\_NULL, tag = MPI\_ANY\_TAG, and count = 0.

 $\mathbf{5}$ 

# Chapter 4

# Partitioned Point-to-Point Communication

# 4.1 Introduction

Partitioned communication extends persistent point-to-point communication as defined in Chapter 3. Partitioned communication operations are matched based on the order in which the local initialization calls are performed. Partitioned communication is "partitioned" because it allows for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation.

Advice to users. The techniques of partitioned communication were known as "finepoints" before their adoption into the MPI standard. We refer the interested reader to the original literature describing the design goals, functioning, initial implementation and performance improvements [28, 29]. (*End of advice to users.*)

 $^{24}$ 

Partitioned communication operations use a persistent communication style that involves a sequence of start and test or wait operations. For this sequence, partitioned communications use MPI\_START or MPI\_STARTALL calls and completion mechanisms (MPI\_TEST or MPI\_WAIT). Partitioned communication is different in three fundamental ways from persistent point-to-point operations in MPI. First, partitioned communication allows additional partitioned test function calls that can expose partial completion of the operation. Second, partitioned communication may perform all of the initialization required to enable data transfer as early as its initialization phase. Third, partitioned communication allows for MPI to be independently notified of multiple contributions from the send-side to a single data buffer of a single MPI message.

*Rationale.* The rationale behind having different initialization behavior allowed for partitioned communication as opposed to persistent point-to-point communication is to enable flexibility and optimization possibilities in implementations. Buffer setup can occur in the partitioned communication initialization functions (see Section 4.2.1). However, such negotiation can be deferred until data is to be moved between two processes. This means that partitioned communication can lazily negotiate as late as testing for completion of the operation on the first iteration of a sequence of partitioned communication start and test or wait operations. Matching still occurs as if matching happened at the partitioned communication initialization functions as noted in the function descriptions. (End of rationale.)

CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

1 2

31

32

33

34

35

36

37

38 39

40

41

42 43

44

45

46

47

48

# 4.2 Semantics of Partitioned Point-to-Point Communication

MPI guarantees certain general properties of partitioned point-to-point communication
 <sup>4</sup> progress, which are described in this section.

Persistent communications use opaque MPI\_REQUEST objects as described in Section 3. Partitioned communication uses these same semantics for MPI\_REQUEST objects.

Partitioned communication provides fine-grained transfers on either or both sides of a 7 send-receive operation described by requests. Persistent communication semantics are ideal 8 for partitioned communication: they provide MPI\_PSEND\_INIT and MPI\_PRECV\_INIT 9 functions that allow partitioned communication setup to occur prior to message transfers. 10 Partitioned communication initialization functions are local. The partitioned communica-11 tion initialization includes inputs on the number of user-visible partitions on the send-side 12and receive-side, which may differ. Valid partitioned communication operations must have 13 one or more partitions specified. 14

Once an MPI\_PSEND\_INIT call has been made, the user may start the operation with a call to a starting procedure and complete the operation with a number of MPI\_PREADY calls equal to the requested number of send partitions followed by a call to a completing procedure. A call to MPI\_PREADY notifies the MPI library that a specified portion of the data buffer (a specific partition) is ready to be sent. Notification of partial completion can be done via fine-grained MPI\_PARRIVED calls at the receiver before a final MPI\_TEST/

MPI\_WAIT on the request itself; the latter represents overall operation completion upon success. A full set of methods for starting and completing partitioned communication is given in the following sections.

- Advice to users. Having a large number of receiver-side partitions can increase over heads as the completion mechanism may need to work with finer-grained notifications.
   Using a small number of receiver-side partitions may provide higher performance.
- A large number of sender-side partitions may be aggregated by an MPI implementation, making performance concerns of a large number of sender-side partitions potentially less impactful than receiver-side granularity. (*End of advice to users.*)
  - Advice to implementors. It is expected that an MPI implementation will attempt to balance latency and aggregation for data transfers for the requested partition counts on the sender-side and receiver-side to allow optimization for different hardware. A high quality implementation may perform significant optimizations to enhance performance in this way; they may, for example, resize the data transfers of the partitions to combine partitions in fractional partition sizes (e.g., 2.5 partitions in a single data transfer). (End of advice to implementors.)

Example 4.1 shows a simple partitioned transfer in which the sender-side and receiverside partitioning is identical in partition count.

**Example 4.1** Simple partitioned communication example.

```
#include "mpi.h"
#define PARTITIONS 8
#define COUNT 5
int main(int argc, char *argv[])
{
```

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

```
double message[PARTITIONS*COUNT];
MPI_Count partitions = PARTITIONS;
int source = 0, dest = 1, tag = 1, flag = 0;
int myrank, i;
int provided;
MPI_Request request;
MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
if (provided < MPI_THREAD_SERIALIZED)
   MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
ſ
   MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
             MPI_COMM_WORLD, MPI_INFO_NULL, &request);
   MPI_Start(&request);
   for(i = 0; i < partitions; ++i)</pre>
   {
      /* compute and fill partition #i, then mark ready: */
      MPI_Pready(i, request);
   }
   while(!flag)
   {
      /* do useful work #1 */
      MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
      /* do useful work #2 */
   }
   MPI_Request_free(&request);
}
else if (myrank == 1)
{
   MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
             MPI_COMM_WORLD, MPI_INFO_NULL, &request);
   MPI_Start(&request);
   while(!flag)
   {
      /* do useful work #1 */
      MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
      /* do useful work #2 */
   }
   MPI_Request_free(&request);
}
MPI_Finalize();
return 0;
```

Rationale. Partitioned communication is designed to provide opportunities for MPI <sup>46</sup> implementations to optimize data transfers. MPI is free to choose how many transfers <sup>47</sup> to do within a partitioned communication send independent of how many partitions <sup>48</sup>

}

are reported as ready to MPI through MPI\_PREADY calls. Aggregation of partitions is permitted but not required. Ordering of partitions is permitted but not required. A naive implementation can simply wait for the entire message buffer to be marked ready before any transfer(s) occur and could wait until the completion function is called on a request before transferring data. However, this modality of communication gives MPI implementations far more flexibility in data movement than non-partitioned communications. (*End of rationale.*)

7 8 9

10

1

 $\mathbf{2}$ 

3

4

5

6

## 4.2.1 Communication Initialization and Starting with Partitioning

Initialization of partitioned communication operations use the initialization calls described below. Subsequent to initialization, MPI\_START/MPI\_STARTALL are used as the first indication to MPI that a message transfer will occur. For send-side operations, neither initializing nor starting the operation enables transfer of any part of the user buffer. Freeing or canceling a partitioned communication request that is active (i.e., initialized and started) and not completed is erroneous. After the partitioned communication operation is started, individual partitions of a message are indicated as ready to be sent by MPI via the MPI\_PREADY function, described below.

18 19 20

21

35

36

37

38

39

MPI\_PSEND\_INIT(buf, partitions, count, datatype, dest, tag, comm, info, request)

22	IN	buf	initial address of send buffer (choice)
23	IN	partitions	number of partitions (non-negative integer)
24 25 26	IN	count	number of elements sent per partition (non-negative integer)
27	IN	datatype	type of each element (handle)
28	IN	dest	rank of destination (integer)
29 30	IN	tag	message tag (integer)
31	IN	comm	communicator (handle)
32	IN	info	info argument (handle)
33 34	OUT	request	communication request (handle)

### C binding

```
<sup>40</sup> Fortran 2008 binding
```

 $^{41}$ MPI\_Psend\_init(buf, partitions, count, datatype, dest, tag, comm, info, 42request, ierror) 43 TYPE(\*), DIMENSION(..), INTENT(IN) :: buf 44INTEGER, INTENT(IN) :: partitions, dest, tag 45INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 46TYPE(MPI\_Datatype), INTENT(IN) :: datatype 47 TYPE(MPI\_Comm), INTENT(IN) :: comm 48 TYPE(MPI\_Info), INTENT(IN) :: info

TYPE(MPI_Request), INTENT(OUT) :: request	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
Fortran binding	3
For train binding	4
MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO,	
	5
REQUEST, IERROR)	f
<type> BUF(*)</type>	
	7
INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR	
	8
INTEGER(KIND=MPI_COUNT_KIND) COUNT	c

MPI\_PSEND\_INIT creates a partitioned communication request and binds to it all the arguments of a partitioned send operation. Matching follows the same MPI matching rules as for point-to-point communication (see Chapter 3) with communicator, tag, and source dictating message matching. In the event that the communicator, tag, and source do not uniquely identify a message, the order in which partitioned communication *initialization* calls are made is the order in which they will eventually match. This operation can only match with partitioned communication initialization operations, therefore it is required to be matched with a corresponding MPI\_PRECV\_INIT call. Partitioned communication initialization calls are local. It is erroneous to provide a partitions value  $\leq 0$ . Send-side and receive-side buffers must be identical in size.

Advice to implementors. Unlike MPI\_SEND\_INIT, MPI\_PSEND\_INIT can be matched as early as the initialization call. Also, unlike MPI\_SEND\_INIT, MPI\_PSEND\_INIT takes an info argument. (End of advice to implementors.)

MPI_PRECV	/_INIT(buf,	partitions,	count,	datatype,	source,	tag,	comm,	info,	request)
									,

IN	buf	initial address of recv buffer (choice)	28
IN	partitions	number of partitions (non-negative integer)	29
IN	count	number of elements received per partition (non-negative integer)	30 31 32
IN	datatype	type of each element (handle)	33
IN	source	rank of source (integer)	34
IN	tag	message tag (integer)	35
IN	comm	communicator (handle)	36 37
IN	info	info argument (handle)	38
Ουτ	request	communication request (handle)	39
		······································	40

#### C binding

int MPI\_Precv\_init(void \*buf, int partitions, MPI\_Count count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request)

Fortran 2008 binding

MPI\_Precv\_init(buf, partitions, count, datatype, source, tag, comm, info, request, ierror)

107

11

1213

14

15

16

17

18

19

20

21

22

27

41

42

43

44

4546

47

```
1
          TYPE(*), DIMENSION(...), INTENT(IN) :: buf
\mathbf{2}
          INTEGER, INTENT(IN) :: partitions, source, tag
3
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
4
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
          TYPE(MPI_Comm), INTENT(IN) :: comm
6
          TYPE(MPI_Info), INTENT(IN) :: info
7
          TYPE(MPI_Request), INTENT(OUT) :: request
8
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     Fortran binding
10
     MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, SOURCE, TAG, COMM, INFO,
11
                    REQUEST, IERROR)
12
          <type> BUF(*)
13
          INTEGER PARTITIONS, DATATYPE, SOURCE, TAG, COMM, INFO, REQUEST, IERROR
14
          INTEGER(KIND=MPI_COUNT_KIND) COUNT
15
16
           Rationale. The info argument is provided in order to support per-operation imple-
17
18
           mentation-defined info keys. (End of rationale.)
19
          MPI_PRECV_INIT creates a partitioned communication receive request and binds to it
20
     all the arguments of a partitioned receive operation. This operation can only match with
21
     partitioned communication initialization operations, therefore the MPI library is required to
22
     match MPI_PRECV_INIT calls only with a corresponding MPI_PSEND_INIT call. Matching
23
     follows the same MPI matching rules as for point-to-point communication (see Chapter 3)
^{24}
     with communicator, tag, and source dictating message matching. In the event that the
25
     communicator, tag, and source do not uniquely identify a message, the order in which
26
     partitioned communication initialization calls are made is the order in which they will
27
     eventually match. Partitioned communication initialization calls are local. That is,
28
     MPI_PRECV_INIT may return before the operation completes. It is erroneous to provide a
29
     partitions value \leq 0. Wildcards for source and tag are not allowed.
30
^{31}
           Advice to implementors. Unlike MPI_RECV_INIT, MPI_PRECV_INIT may communi-
32
           cate. Also unlike MPI_RECV_INIT, MPI_PRECV_INIT takes an info argument. (End
33
           of advice to implementors.)
34
35
36
37
     MPI_PREADY(partition, request)
38
       IN
                 partition
                                             partition to mark ready for transfer (non-negative
39
                                             integer)
40
41
       INOUT
                 request
                                             partitioned communication request (handle)
42
43
     C binding
44
     int MPI_Pready(int partition, MPI_Request request)
45
     Fortran 2008 binding
46
     MPI_Pready(partition, request, ierror)
47
          INTEGER, INTENT(IN) :: partition
48
```

 $\mathbf{2}$ 

MPI\_PREADY is a send-side call that indicates that a given partition is ready to be transferred. It is erroneous to use MPI\_PREADY on any request object that does not correspond to a partitioned send operation. The partitioning is defined by the MPI\_PSEND\_INIT call. Partition numbering starts at zero and ranges to one less than the number of partitions declared in the MPI\_PSEND\_INIT call. Specifying a partition number that is equal to or larger than the number of partitions is erroneous. After a call to MPI\_START/MPI\_STARTALL, all partitions associated with that operation are inactive. A call to MPI\_PREADY marks the indicated partition as active. Calling MPI\_PREADY on an active partition is erroneous.

MPI\_PREADY\_RANGE(partition\_low, partition\_high, request)

IN	partition_low	lowest partition ready for transfer (non-negative integer)	19 20		
IN	partition_high	highest partition ready for transfer (non-negative integer)	21 22 23		
INOUT	request	partitioned communication request (handle)	23 24 25		
	C binding int MPI_Pready_range(int partition_low, int partition_high, MPI_Request request)				
<pre>Fortran 2008 binding MPI_Pready_range(partition_low, partition_high, request, ierror) INTEGER, INTENT(IN) :: partition_low, partition_high TYPE(MPI_Request), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
<pre>Fortran binding MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR) INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR A call to MPI_PREADY_RANGE has the same effect as calls to MPI_PREADY, executed for i=partition_low,, partition_high, in some arbitrary order.</pre>					
		ie same rules as those for MPI_PREADY calls.	40 41 42 43 44		

	110	CHAPTER 4.	PARTITIONED POINT-TO-POINT COMMUNICATION
1	MPI_PREA	ADY_LIST(length, ar	ray_of_partitions, request)
2 3	IN	length	list length (integer)
4	IN	array_of_partitions	array of partitions (array of non-negative integers)
5 6	INOUT	request	partitioned communication request (handle)
7 8 9 10	C binding int MPI_P	-	<pre>ength, const int array_of_partitions[], request)</pre>
11 12 13 14 15	MPI_Pread INTEG TYPE(	ER, INTENT(IN) : MPI_Request), IN	rray_of_partitions, request, ierror) : length, array_of_partitions(length) TENT(IN) :: request TENT(OUT) :: ierror
16 17 18 19		Y_LIST(LENGTH, A	RRAY_OF_PARTITIONS, REQUEST, IERROR) _OF_PARTITIONS(*), REQUEST, IERROR
20 21 22 23 24	MPI_PREA	DY, executed for the _of_partitions[cours]	LIST has the same effect as calls to he partitions specified in the range $array_of_partitions[0]$ $nt - 1$ ] of the array_of_partitions, executed in some arbitrary _LIST follow the same rules as those for MPI_PREADY calls.
25	4.2.2 Cor	mmunication Comp	letion under Partitioning
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	communication the sender subsequent the user can partitioned does not in The control of the transformation of the transformation can be called a sender the transformation called a sender the transformation can be called a sender the transfo	ation operation. The is now free to call by MPI_PREADY, M an safely free the part of operation. For the indicate that the part ompletion of a part that the receive buff eption of the receive an be used to determ to the receive buff	MPI_TEST (and variants) are used to complete a partitioned the completion of a partitioned send operation indicates that MPI_START/MPI_STARTALL to restart the operation and MPI_PREADY_RANGE or MPI_PREADY_LIST. Alternatively, rititioned communication request after the completion of the sending process, completion of the partitioned send operation titions of the message have all been received. itioned receive operation through MPI_WAIT or MPI_TEST for contains all of the partitions. A function for probing the buffer is provided by MPI_PARRIVED. The MPI_PARRIVED nine if the message data for the indicated partition has been er. Upon success, the receiver becomes free to access the any others that previously completed for that operation).

MPI_PARRIVED(request, partition, flag)				
IN	request	partitioned communication request (handle)	2	
	1044000	partitioned communication request (name)	3	
IN	partition	partition to be tested (non-negative integer)	4	
OUT	flag	true if operation completed on the specified partition,	5	
		false if not (logical)	6	
			7	
C binding	<b>y</b>		8	
		est, int partition, int *flag)	9	
1		550, 110 parororon, 110 (1148)	10	
Fortran 2008 binding			11	
MPI_Parrived(request, partition, flag, ierror)			12	
TYPE(MPI_Request), INTENT(IN) :: request			13	
INTEG	ER, INTENT(IN) :: partit:	ion	14	
LOGIC	AL, INTENT(OUT) :: flag		15	
INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
Fortran b	vinding		17	
	0		18	
	VED (REQUEST, PARTITION, I	-	19	
	INTEGER REQUEST, PARTITION, IERROR			
LUGIC	LOGICAL FLAG			
The function MPI_PARRIVED can be used to test partial completion of partitioned 22				

receive operations. A call to MPI\_PARRIVED can be used to test partial completion of partitioned receive operations. A call to MPI\_PARRIVED on an active partitioned communication request returns flag = true if the operation identified by request for the specified partition is complete. The request is not marked as complete/inactive by this procedure. A subsequent call to an MPI completing procedure (e.g., MPI\_TEST/MPI\_WAIT) is required to complete the operation, as described in Chapter 3. MPI\_PARRIVED may be called multiple times for a partition. MPI\_PARRIVED may be called with a null or inactive request argument. In either case, the operation returns with flag = true. Calling MPI\_PARRIVED on a request that does not correspond to a partitioned receive operation is erroneous.

 $^{24}$ 

 $^{31}$ 

Repeated calls to MPI\_PARRIVED with the same request and partition arguments will eventually return flag = true if the corresponding partitioned send operation has been started and all send partitions have been marked as ready. For additional information on MPI *progress* see Section 3.7.4.

Advice to implementors. A high quality implementation will eventually return flag = true from MPI\_PARRIVED after all of the corresponding MPI\_PREADY calls have been made for a receive-side partition, even if other send partitions are not yet marked as ready. (*End of advice to implementors.*)

#### 4.2.3 Semantics of Communications in Partitioned Mode

The semantics of nonblocking partitioned communication are defined by suitably extending the definitions in Section 3.5.

Interpretation of count and datatype for partitioned communication Partitioned communication uses the count and datatype arguments in the partitioned communication initialization functions to describe a single partition. The argument partitions specifies how many

CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

equal partitions of a number (count) of objects of datatypes make up the entire buffer to
 be transferred in the partitioned communication. As partitioned communication describes
 many partitions, using absolute displacements in datatypes (e.g., MPI\_BOTTOM) is not
 supported. Partitions are contiguous in memory, there is no padding in between them.
 Once a partitioned send operation is started, each partition must be marked as ready using
 MPI\_PREADY and the operation must be completed using a completion function, such as
 MPI\_TEST or MPI\_WAIT.

8 9

10

11

12

**Order** Matching follows the same MPI matching rules as for point-to-point communication (see Chapter 3) with communicator, tag, and source dictating message matching. In the event that the communicator, tag, and source do not uniquely identify the message, the order in which partitioned communication initialization calls are made is the order in which they will eventually match.

13 14 15

16 17

18

# 4.3 Partitioned Communication Examples

This section provides concrete examples of the utility of partitioned communication in realistic settings.

19 20 21

### 4.3.1 Partition Communication with Threads/Tasks Using OpenMP 4.0 or later

The equal partitioning on send-side and receive-side in Example 4.1 is shown using threads. In this case, the receive-side uses the same number of partitions as the send-side like in the previous example, but this example uses multiple threads on the send-side. Note that the MPI\_PSEND\_INIT and MPI\_PRECV\_INIT functions match each other like in the previous example.

```
Example 4.2 Equal partitioning on send-side and receive-side using threads.
28
29
     #include "mpi.h"
30
     #define NUM_THREADS 8
31
     #define PARTITIONS 8
32
     #define PARTLENGTH 16
33
     int main(int argc, char *argv[]) /* same send/recv partitioning */
34
     {
35
       double message[PARTITIONS*PARTLENGTH];
36
       int partitions = PARTITIONS;
37
       int partlength = PARTLENGTH;
38
       int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
39
       int myrank;
40
       int provided;
41
       MPI_Request request;
42
       MPI_Info info = MPI_INFO_NULL;
43
       MPI_Datatype xfer_type;
44
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
45
       if (provided < MPI_THREAD_MULTIPLE)
46
          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
47
       MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
48
```

```
MPI_Type_contiguous(partlength, MPI_DOUBLE, &xfer_type);
  MPI_Type_commit(&xfer_type);
  if (myrank == 0)
  {
     MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
          info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);
     #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
     for (int i=0; i<partitions; i++)</pre>
     {
        /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, request);
     }
     while(!flag)
     {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
  else if (myrank == 1)
  {
     MPI_Precv_init(message, partitions, count, xfer_type, source, tag,
           info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);
     while(!flag)
     {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
  MPI_Finalize();
  return 0;
}
```

# 4.3.2 Send-only Partitioning Example with Tasks and OpenMP version 4.0 or later

The previous example is tailored specifically for send-side partitioning using threads. This is an example where parallel task producers produce input to part of an overall buffer; they complete in any order and contribute to the overall buffer.

 $\label{eq:Example 4.3} {\rm \ Parallel\ task\ producers\ for\ partitioned\ communication\ using\ threads}.$ 

```
#include "mpi.h"
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8 9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39 40

 $41 \\ 42$ 

43 44

45

 $46 \\ 47$ 

CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

```
1
     #define NUM_THREADS 8
\mathbf{2}
     #define NUM_TASKS 64
3
     #define PARTITIONS NUM_TASKS
4
     #define PARTLENGTH 16
5
     #define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
6
     int main(int argc, char *argv[]) /* send-side partitioning */
7
     {
8
       double message[MESSAGE_LENGTH];
9
       int send_partitions = PARTITIONS,
10
            send_partlength = PARTLENGTH,
11
            recv_partitions = 1,
12
            recv_partlength = PARTITIONS*PARTLENGTH;
13
       int count = 1, source = 0, dest = 1, tag = 1, flag = 0;
14
       int myrank;
15
       int provided;
16
       MPI_Request request;
17
       MPI_Info info = MPI_INFO_NULL;
18
       MPI_Datatype send_type;
19
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
20
       if (provided < MPI_THREAD_MULTIPLE)</pre>
21
           MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
22
       MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
23
       MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
24
       MPI_Type_commit(&send_type);
25
26
       if (myrank == 0)
27
       {
28
           MPI_Psend_init(message, send_partitions, count, send_type, dest, tag,
29
                     info, MPI_COMM_WORLD, &request);
30
           MPI_Start(&request);
31
32
           #pragma omp parallel shared(request) num_threads(NUM_THREADS)
33
           {
34
              #pragma omp single
35
              {
36
                 /* single thread creates 64 tasks to be executed by 8 threads */
37
                 for (int partition_num=0;partition_num<NUM_TASKS;partition_num++)</pre>
38
                 {
39
                    #pragma omp task firstprivate(partition_num)
40
                    {
41
                       /* compute and fill partition #partition_num, then mark
42
                       ready: */
43
                       /* buffer is filled in arbitrary order from each task */
44
                       MPI_Pready(partition_num, request);
45
                    } /*end task*/
46
                 } /* end for */
47
              } /* end single */
48
```

```
} /* end parallel */
     while(!flag)
     {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
  else if (myrank == 1)
  {
     MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
               source, tag, info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);
     while(!flag)
     ſ
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
  MPI_Finalize();
  return 0;
}
```

# 4.3.3 Send and Receive Partitioning Example with OpenMP version 4.0 or later

This example demonstrates receive-side partial completion notification using more than one partition per receive-side thread. It uses a naive flag based method to test for multiple completed partitions per thread. Note that this means that some threads may be busy polling for completion of assigned partitions when partitions are available to work on that were not assigned to the polling threads in this example. More advanced work stealing methods could be employed for greater efficiency. Like previous examples, it also demonstrates send-side production of input to part of an overall buffer. This example also uses different send-side and receive-side partitioning.

**Example 4.4** Partitioned communication receive-side partial completion.

```
#include "mpi.h"
#define NUM_THREADS 64
#define PARTITIONS NUM_THREADS
#define PARTLENGTH 16
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
int main(int argc, char *argv[]) /* send-side partitioning */
{
```

 $^{31}$ 

```
1
       double message[MESSAGE_LENGTH];
\mathbf{2}
       int send_partitions = PARTITIONS,
3
           send_partlength = PARTLENGTH,
4
           recv_partitions = PARTITIONS*2,
5
           recv_partlength = PARTLENGTH/2;
6
       int source = 0, dest = 1, tag = 1, flag = 0;
7
       int myrank;
8
       int provided;
9
       MPI_Request request;
10
       MPI_Info info = MPI_INFO_NULL;
11
       MPI_Datatype send_type;
12
       MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
13
       if (provided < MPI_THREAD_MULTIPLE)</pre>
14
          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
15
       MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
16
       MPI_Type_contiguous(send_partlength, MPI_DOUBLE, &send_type);
17
       MPI_Type_commit(&send_type);
18
19
       if (myrank == 0)
20
       {
21
          MPI_Psend_init(message, send_partitions, 1, send_type, dest, tag,
22
                     info, MPI_COMM_WORLD, &request);
23
          MPI_Start(&request);
24
          #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
25
           for (int i=0; i<send_partitions; i++)</pre>
26
           {
27
              /* compute and fill partition #i, then mark ready: */
28
              MPI_Pready(i, request);
29
           }
30
          while(!flag)
31
           {
32
              /* Do useful work */
33
              MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
34
              /* Do useful work */
35
           }
36
          MPI_Request_free(&request);
37
       }
38
       else if (myrank == 1)
39
       {
40
          MPI_Precv_init(message, recv_partitions, recv_partlength, MPI_DOUBLE,
41
                     source, tag, info, MPI_COMM_WORLD, &request);
42
          MPI_Start(&request);
43
          #pragma omp parallel for shared(request) num_threads(NUM_THREADS)
44
          for (int j=0; j<recv_partitions; j+=2)</pre>
45
           {
46
              int part1_complete = 0;
47
              int part2_complete = 0;
48
```

```
while(part1_complete == 0 || part2_complete == 0)
        {
           /* test partition #j and #j+1 */
           MPI_Parrived(request, j, &flag);
           if(flag && part1_complete == 0)
           {
              part1_complete++;
              /* Do work using partition j data */
           }
           if (j+1 < recv_partitions) {</pre>
              MPI_Parrived(request, j+1, &flag);
              if(flag && part2_complete == 0)
              {
                 part2_complete++;
                 /* Do work using partition j+1 */
              }
           }
           else {
               part2_complete++;
           }
        }
     }
     while(!flag)
     {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* Do useful work */
     }
     MPI_Request_free(&request);
  }
  MPI_Finalize();
  return 0;
}
```

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

## CHAPTER 4. PARTITIONED POINT-TO-POINT COMMUNICATION

# Chapter 5

# Datatypes

Basic datatypes were introduced in Section 3.2.2 and in Section 3.3. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

 $^{24}$ 

 $45 \\ 46$ 

# 5.1 Derived Datatypes

Up to here, all point-to-point communications have involved only buffers containing a sequence of identical basic datatypes. This is too constraining on two accounts. One often wants to pass messages that contain values with different datatypes (e.g., an integer count, followed by a sequence of real numbers); and one often wants to send noncontiguous data (e.g., a sub-block of a matrix). One solution is to pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site. This has the disadvantage of requiring additional memory-to-memory copy operations at both sites, even when the communication subsystem has scatter-gather capabilities. Instead, MPI provides mechanisms to specify more general, mixed, and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides.

The general mechanisms provided here allow one to transfer directly, without copying, objects of various shapes and sizes. It is not assumed that the MPI library is cognizant of the objects declared in the host language. Thus, if one wants to transfer a structure, or an array section, it will be necessary to provide in MPI a definition of a communication buffer that mimics the definition of the structure or array section in question. These facilities can be used by library designers to define communication functions that can transfer objects defined in the host language—by decoding their definitions as available in a symbol table or a dope vector. Such higher-level communication functions are not part of MPI.

More general communication buffers are specified by replacing the basic datatypes that have been used so far with derived datatypes that are constructed from basic datatypes using the constructors described in this section. These methods of constructing derived datatypes can be applied recursively.

A general datatype is an opaque object that specifies two things:

- A sequence of basic datatypes
- A sequence of integer (byte) displacements

The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. We call such a pair of sequences (or sequence of pairs) a **type map**. The sequence of basic datatypes (displacements ignored) is the **type signature** of the datatype.

6

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

7 8

9

10

11

 $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$ 

be such a type map, where  $type_i$  are basic types, and  $disp_i$  are displacements. Let

 $Typesig = \{type_0, \dots, type_{n-1}\}$ 

<sup>12</sup> be the associated type signature. This type map, together with a base address **buf**, specifies <sup>13</sup> a communication buffer: the communication buffer that consists of n entries, where the <sup>14</sup> *i*-th entry is at address **buf** +  $disp_i$  and has type  $type_i$ . A message assembled from such a <sup>15</sup> communication buffer will consist of n values, of the types defined by Typesig.

<sup>16</sup> Most datatype constructors have replication count or block length arguments. Allowed
 <sup>17</sup> values are non-negative integers. If the value is zero, no elements are generated in the type
 <sup>18</sup> map and there is no effect on datatype bounds or extent.

<sup>19</sup> We can use a handle to a general datatype as an argument in a send or receive operation, <sup>20</sup> instead of a basic datatype argument. The operation MPI\_SEND(buf, 1, datatype,...) will <sup>21</sup> use the send buffer defined by the base address buf and the general datatype associated <sup>22</sup> with datatype; it will generate a message with the type signature determined by the datatype <sup>23</sup> argument. MPI\_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base <sup>24</sup> address buf and the general datatype associated with datatype.

<sup>25</sup> General datatypes can be used in all send and receive operations. We discuss, in <sup>26</sup> Section 5.1.11, the case where the second argument **count** has value > 1.

<sup>27</sup> The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, <sup>28</sup> and are predefined. Thus, MPI\_INT is a predefined handle to a datatype with type map <sup>29</sup> {(int,0)}, with one entry of type int and displacement zero. The other basic datatypes <sup>30</sup> are similar. <sup>31</sup> The **axtent** of a datatype is defined to be the span from the first bute to the last byte

The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

 $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$ 

then

37 38

32

33

34

35 36

 $lb(Typemap) = \min_{j} disp_{j},$   $ub(Typemap) = \max_{j} (disp_{j} + \text{sizeof}(type_{j})) + \epsilon, \text{ and}$ extent(Typemap) = ub(Typemap) - lb(Typemap).(5.1)

<sup>42</sup> If  $type_j$  requires alignment to a byte address that is a multiple of  $k_j$ , then  $\epsilon$  is the least <sup>43</sup> non-negative increment needed to round extent(Typemap) to the next multiple of  $\max_j k_j$ . <sup>45</sup> In Fortran, it is implementation dependent whether the MPI implementation computes <sup>46</sup> the alignments  $k_j$  according to the alignments used by the compiler in common blocks, <sup>47</sup> SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE <sup>48</sup> nor BIND(C). The complete definition of **extent** is given by Equation 5.1 Section 5.1.

Let

**Example 5.1** Assume that  $Type = \{(double, 0), (char, 8)\}$  (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 5.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. In Fortran, structures can be expressed with several language features, e.g., common blocks, SEQUENCE derived types, or BIND(C) derived types. The compiler may use different alignments, and therefore, it is recommended to use MPI\_TYPE\_CREATE\_RESIZED for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 5.1.6 and in Section 19.1.15. (*End of rationale.*)

### 5.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI\_TYPE\_CREATE\_HVECTOR, MPI\_TYPE\_CREATE\_HINDEXED, MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK, MPI\_TYPE\_CREATE\_STRUCT, and MPI\_GET\_ADDRESS accept arguments of type INTEGER(KIND=MPI\_ADDRESS\_KIND), wherever arguments of type MPI\_Aint are used in C. For Fortran compilers that do not support the Fortran 90 KIND notation, and where addresses are 64 bits whereas default INTEGERs are 32 bits, these arguments will be of type INTEGER\*8 (assuming the Fortran compiler accepts the common extension of INTEGER\*8 for eight-byte integers).

For the large count versions of three datatype constructors with explicit addresses, MPI\_TYPE\_CREATE\_HINDEXED, MPI\_TYPE\_CREATE\_HINDEXED\_BLOCK, and MPI\_TYPE\_CREATE\_STRUCT, absolute addresses shall not be used to specify byte displacements since the parameter is of type MPI\_COUNT instead of type MPI\_AINT.

### 5.1.2 Datatype Constructors

**Contiguous** The simplest datatype constructor is MPI\_TYPE\_CONTIGUOUS which allows replication of a datatype into contiguous locations.

### MPI\_TYPE\_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count (non-negative integer)	40	
IN	oldtype	old datatype (handle)	41	
OUT	newtype	new datatype (handle)	42	
001	newtype	new databype (number)	43	
44				
C binding			45	

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11

12

13

14

15

16

17 18

19

20

21

22

23 24

25

26

27

28

29 30

31

32 33

34

35

36 37 38

39

46

12	<pre>int MPI_Type_contiguous_c(MPI_Count count, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>
3 4 5 6 7 8 9	<pre>Fortran 2008 binding MPI_Type_contiguous(count, oldtype, newtype, ierror)     INTEGER, INTENT(IN) :: count     TYPE(MPI_Datatype), INTENT(IN) :: oldtype     TYPE(MPI_Datatype), INTENT(OUT) :: newtype     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
10 11 12 13 14	<pre>MPI_Type_contiguous(count, oldtype, newtype, ierror) !(_c)     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count     TYPE(MPI_Datatype), INTENT(IN) :: oldtype     TYPE(MPI_Datatype), INTENT(OUT) :: newtype     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
15 16 17 18	Fortran binding MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
19 20 21	<b>newtype</b> is the datatype obtained by concatenating <b>count</b> copies of <b>oldtype</b> . Concatenation is defined using <i>extent</i> as the size of the concatenated copies.
21 22 23	<b>Example 5.2</b> Let oldtype have type map $\{(double, 0), (char, 8)\}$ , with extent 16, and let count = 3. The type map of the datatype returned by newtype is
24 25	$\{(\texttt{double}, 0), (\texttt{char}, 8), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{double}, 32), (\texttt{char}, 40)\};$
26 27	i.e., alternating double and char elements, with displacements 0, 8, 16, 24, 32, 40.
28	In general, assume that the type map of <b>oldtype</b> is
29 30	$\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$
31 32	with extent $ex$ . Then newtype has a type map with count $\cdot$ n entries defined by:
33 34	$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots, (type_{n-1}, disp_{n-1}), \dots, (type_{n-1}, disp_{n-1}), \dots, (type_{n-1}, disp_{n-1}), \dots, (type_{n-1}, disp_{n-1}), \dots, \dots,$
35 36	$\ldots, (type_0, disp_0 + ex \cdot (count - 1)), \ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.$
37 38 39 40 41	Vector The function MPI_TYPE_VECTOR is a more general constructor that allows repli- cation of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.
42 43 44	
45	
$\frac{46}{47}$	
48	

MPI_TYF	PE_VECTOR(count, block	klength, stride, oldtype, newtype)			
IN	count	number of blocks (non-negative integer)			
IN	blocklength	number of elements in each block (non-negative integer)			
IN	stride	number of elements between start of each block (integer)			
IN	oldtype	old datatype (handle)			
OUT	newtype	new datatype (handle)			
001	newtype	new datatype (nandre)			
C bindi	ng				
	_Type_vector(int coun	t, int blocklength, int stride, Ldtype, MPI_Datatype *newtype)			
int MPT	Type vector c(MPI Co	unt count, MPI_Count blocklength,			
±	01	le, MPI_Datatype oldtype, MPI_Datatype *newtype)			
Dontero					
	2008 binding	length, stride, oldtype, newtype, ierror)			
• -		ount, blocklength, stride			
	E(MPI_Datatype), INTE	•			
TYPE	E(MPI_Datatype), INTE	NT(OUT) :: newtype			
INTE	INTEGER, OPTIONAL, INTENT(OUT) :: ierror 22				
INTE TYPE TYPE		NT(OUT) :: newtype			
	E_VECTOR (COUNT, BLOCK	LENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR) TH, STRIDE, OLDTYPE, NEWTYPE, IERROR			
	6. A call to MPI_TYPE_V	that oldtype has type map {(double,0), (char,8)}, with /ECTOR(2, 3, 4, oldtype, newtype) will create the datatype			
{(d	ouble, 0), (char, 8), (dou)	ble, 16), (char, 24), (double, 32), (char, 40),			
(do	$(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104) \}.$				
	two blocks with three copetween the the start of ea	pies each of the old type, with a stride of 4 elements $(4 \cdot 16$ ach block.			
Exampl datatype		YPE_VECTOR(3, 1, -2, oldtype, newtype) will create the			
{(d	louble,0),(char.8),(dou	$ble, -32), (char, -24), (double, -64), (char, -56)\}.$			
	, , , , , , , , , , , , , , , , , , , ,				

1	In general, assume that oldtype has type map,						
2 3	$\{(ty)$	$pe_0, disp_0), \ldots, (type_{n-1}, disp_n)$	$-1)\},$				
4 5 6	with extent $ex$ . Let bl be the blocklength. The newly created datatype has a type map with count $\cdot$ bl $\cdot$ n entries:						
7 8	$\{(ty)$	$\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}),\$					
9	(typ	$e_0, disp_0 + ex), \dots, (type_{n-1}, d)$	$isp_{n-1} + ex), \ldots,$				
10 11	(typ	$(type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),$					
12 13	(typ	$e_0, disp_0 + stride \cdot ex), \dots, (typ)$	$e_{n-1}, disp_{n-1} + stride \cdot ex), \ldots,$				
14 15	(typ	$e_0, disp_0 + (stride + bl - 1) \cdot ex$	$(type_{n-1}, disp_{n-1} + (stride + bl - 1) \cdot ex), \ldots,$				
1617	(typ	$e_0, disp_0 + stride \cdot (count - 1) \cdot$	$ex),\ldots,$				
18 19	(typ	$e_{n-1}, disp_{n-1} + stride \cdot (count + stride)$	$(-1) \cdot ex), \ldots,$				
20 21	(typ	$(type_0, disp_0 + (stride \cdot (count - 1) + bl - 1) \cdot ex), \dots,$					
22 22 23	$(type_{n-1}, disp_{n-1} + (stride \cdot (count - 1) + bl - 1) \cdot ex)\}.$						
24 25 26	A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), where n is an arbitrary integer value.						
27 28 29 30 31 32	Hvector The function MPI_TYPE_CREATE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 5.1.14. (H stands for "heterogeneous").						
33 34	MPI_TYP	E_CREATE_HVECTOR(count,	blocklength, stride, oldtype, newtype)				
35	IN	count	number of blocks (non-negative integer)				
36 37	IN	blocklength	number of elements in each block (non-negative integer)				
38 39	IN	stride	number of bytes between start of each block (integer)				
40	IN	oldtype	old datatype (handle)				
41 42	OUT	newtype	new datatype (handle)				
43 44 45 46	C binding int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)						
47 48	int Millighe_cleate_nvector_c(Mil_count count, Mil_count brocklength,						

```
Fortran 2008 binding
                                                                                                       1
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
                                                                                                      2
                                                                                                       3
                 ierror)
     INTEGER, INTENT(IN) :: count, blocklength
                                                                                                       4
     INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
                                                                                                       5
                                                                                                       6
     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                                       7
     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                                       8
                                                                                                      9
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
                                                                                                      10
                 ierror) !(_c)
                                                                                                      11
     INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride
                                                                                                      12
     TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                                      13
     TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                                      14
     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                                      15
                                                                                                      16
Fortran binding
                                                                                                      17
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
                                                                                                      18
                 IERROR)
                                                                                                      19
     INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
     INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
                                                                                                      20
                                                                                                      21
     Assume that oldtype has type map,
                                                                                                      22
                                                                                                      23
      \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\
                                                                                                      24
with extent ex. Let bl be the blocklength. The newly created datatype has a type map with
                                                                                                      25
count \cdot bl \cdot n entries:
                                                                                                      26
                                                                                                      27
      \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1}), \}
                                                                                                      28
                                                                                                      29
      (type_0, disp_0 + ex), \dots, (type_{n-1}, disp_{n-1} + ex), \dots,
                                                                                                      30
                                                                                                      31
      (type_0, disp_0 + (bl - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + (bl - 1) \cdot ex),
                                                                                                      32
                                                                                                      33
      (type_0, disp_0 + \mathsf{stride}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride}), \ldots,
                                                                                                      34
                                                                                                      35
      (type_0, disp_0 + stride + (bl - 1) \cdot ex), \ldots,
                                                                                                      36
      (type_{n-1}, disp_{n-1} + stride + (bl - 1) \cdot ex), \ldots,
                                                                                                      37
                                                                                                      38
      (type_0, disp_0 + stride \cdot (count - 1)), \ldots, (type_{n-1}, disp_{n-1} + stride \cdot (count - 1)), \ldots,
                                                                                                      39
                                                                                                      40
      (type_0, disp_0 + stride \cdot (count - 1) + (bl - 1) \cdot ex), \dots,
                                                                                                      41
                                                                                                      42
      (type_{n-1}, disp_{n-1} + stride \cdot (count - 1) + (bl - 1) \cdot ex)\}.
                                                                                                      43
                                                                                                      44
                                                                                                      45
                                                                                                      46
                                                                                                      47
                                                                                                      48
```

Indexed The function MPI\_TYPE\_INDEXED allows replication of an old datatype into a  $\mathbf{2}$ sequence of blocks (each block is a concatenation of the old datatype), where each block 3 can contain a different number of copies and have a different displacement. All block 4 displacements are multiples of the old type extent. 56 MPI\_TYPE\_INDEXED(count, array\_of\_blocklengths, array\_of\_displacements, oldtype, 7 newtype) 8 9 IN number of blocks—also number of entries in count 10 array\_of\_displacements and array\_of\_blocklengths 11 (non-negative integer) 12IN array\_of\_blocklengths number of elements per block (array of non-negative 13 integers) 14IN array\_of\_displacements displacement for each block, in multiples of oldtype 15(array of integers) 1617 IN oldtype old datatype (handle) 18 OUT newtype new datatype (handle) 19 20C binding 21int MPI\_Type\_indexed(int count, const int array\_of\_blocklengths[], 22 const int array\_of\_displacements[], MPI\_Datatype oldtype, 23MPI\_Datatype \*newtype) 2425int MPI\_Type\_indexed\_c(MPI\_Count count, 26const MPI\_Count array\_of\_blocklengths[], 27const MPI\_Count array\_of\_displacements[], 28MPI\_Datatype oldtype, MPI\_Datatype \*newtype) 29 Fortran 2008 binding 30 MPI\_Type\_indexed(count, array\_of\_blocklengths, array\_of\_displacements,  $^{31}$ oldtype, newtype, ierror) 32 INTEGER, INTENT(IN) :: count, array\_of\_blocklengths(count), 33 array\_of\_displacements(count) 34 TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 35TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 36 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 37 38 MPI\_Type\_indexed(count, array\_of\_blocklengths, array\_of\_displacements, 39 oldtype, newtype, ierror) !(\_c) 40 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count, 41 array\_of\_blocklengths(count), array\_of\_displacements(count) 42TYPE(MPI\_Datatype), INTENT(IN) :: oldtype 43 TYPE(MPI\_Datatype), INTENT(OUT) :: newtype 44INTEGER, OPTIONAL, INTENT(OUT) :: ierror 45 Fortran binding 46 MPI\_TYPE\_INDEXED(COUNT, ARRAY\_OF\_BLOCKLENGTHS, ARRAY\_OF\_DISPLACEMENTS, 47OLDTYPE, NEWTYPE, IERROR) 48

```
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
OLDTYPE, NEWTYPE, IERROR
```

 $\mathbf{2}$ **Example 5.5** Let oldtype have type map  $\{(double, 0), (char, 8)\}$ , with extent 16. Let B =(3, 1) and let D = (4, 0). A call to MPI\_TYPE\_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,  $\{(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104), \}$  $(\texttt{double}, 0), (\texttt{char}, 8)\}.$ That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0. In general, assume that oldtype has type map,  $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$ with extent *ex*. Let B be the array\_of\_blocklengths argument and D be the array\_of\_displacements argument. The newly created datatype has  $n \cdot \sum_{i=0}^{count-1} B[i]$  entries: { $(type_0, disp_0 + \mathsf{D}[0] \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{D}[0] \cdot ex), \ldots,$  $(type_0, disp_0 + (D[0] + B[0] - 1) \cdot ex), \dots,$  $(type_{n-1}, disp_{n-1} + (D[0] + B[0] - 1) \cdot ex), \dots,$  $^{24}$  $(type_0, disp_0 + \mathsf{D}[\mathsf{count-1}] \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{D}[\mathsf{count-1}] \cdot ex), \ldots,$  $(type_0, disp_0 + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex), \dots,$  $(type_{n-1}, disp_{n-1} + (\mathsf{D}[\mathsf{count-1}] + \mathsf{B}[\mathsf{count-1}] - 1) \cdot ex)\}.$ A call to MPI\_TYPE\_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI\_TYPE\_INDEXED(count, B, D, oldtype, newtype) where  $D[j] = j \cdot stride, j = 0, \dots, count - 1,$ and  $B[j] = blocklength, j = 0, \dots, count - 1.$ Hindexed The function MPI\_TYPE\_CREATE\_HINDEXED is identical to MPI\_TYPE\_INDEXED, except that block displacements in array\_of\_displacements are spec-ified in bytes, rather than in multiples of the oldtype extent. 

1 2	MPI_TYPI	E_CREATE_HINDEXED(coun oldtype, newtype)	nt, array_of_blocklengths, array_of_displacements,
3 4 5 6	IN	count	number of blocks—also number of entries in array_of_displacements and array_of_blocklengths (non-negative integer)
7 8	IN	array_of_blocklengths	number of elements in each block (array of non-negative integers)
9	IN	array_of_displacements	byte displacement of each block (array of integers)
10	IN	oldtype	old datatype (handle)
11 12	OUT	newtype	new datatype (handle)
13 14 15 16 17	C binding int MPI_7	Type_create_hindexed(int	<pre>count, const int array_of_blocklengths[], .y_of_displacements[], MPI_Datatype oldtype, .pe)</pre>
18 19 20 21 22	int MPI_7	const MPI_Count arr	<pre>PI_Count count, ray_of_blocklengths[], ray_of_displacements[], we, MPI_Datatype *newtype)</pre>
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39	MPI_Type_ INTEC INTEC TYPE( INTEC MPI_Type_ INTEC TYPE( TYPE)	<pre>SER, INTENT(IN) :: count SER(KIND=MPI_ADDRESS_KIND array_of_displaceme (MPI_Datatype), INTENT(ID (MPI_Datatype), INTENT(OD SER, OPTIONAL, INTENT(OD _create_hindexed(count, a array_of_displaceme SER(KIND=MPI_COUNT_KIND)</pre>	<pre>ents, oldtype, newtype, ierror) , array_of_blocklengths(count) D), INTENT(IN) :: ents(count) N) :: oldtype UT) :: newtype T) :: ierror array_of_blocklengths, ents, oldtype, newtype, ierror) !(_c) , INTENT(IN) :: count, ths(count), array_of_displacements(count) N) :: oldtype UT) :: newtype</pre>
40 41 42 43 44 45	INTEC	CREATE_HINDEXED(COUNT, A ARRAY_OF_DISPLACEME GER COUNT, ARRAY_OF_BLOCH GER(KIND=MPI_ADDRESS_KIN)	ENTS, OLDTYPE, NEWTYPE, IERROR) KLENGTHS(*), OLDTYPE, NEWTYPE, IERROR D) ARRAY_OF_DISPLACEMENTS(*)
46		ne that <b>oldtype</b> has type map	
47 48	$\{(typ)\}$	$(be_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})$	$\{p_{n-1}\}\},$

with extent *ex*. Let B be the array\_of\_blocklengths argument and D be the array\_of\_displacements argument. The newly created datatype has a type map with  $n \cdot$  $\sum_{i=0}^{\text{count}-1} B[i]$  entries:

$$\{(type_0, disp_0 + D[0]), \dots, (type_{n-1}, disp_{n-1} + D[0]), \dots, \\ (type_0, disp_0 + D[0] + (B[0] - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + D[0] + (B[0] - 1) \cdot ex), \dots, \\ (type_0, disp_0 + D[count-1]), \dots, (type_{n-1}, disp_{n-1} + D[count-1]), \dots, \\ (type_0, disp_0 + D[count-1] + (B[count-1] - 1) \cdot ex), \dots, \\ (type_{n-1}, disp_{n-1} + D[count-1] + (B[count-1] - 1) \cdot ex)\}.$$

Indexed\_block This function is the same as MPI\_TYPE\_INDEXED except that the blocklength is the same for all blocks. There are many codes using indirect addressing arising from unstructured grids where the blocksize is always 1 (gather/scatter). The following convenience function allows for constant blocksize and arbitrary displacements.

### MPI\_TYPE\_CREATE\_INDEXED\_BLOCK(count, blocklength, array\_of\_displacements, oldtype, newtype)

IN	count	number of blocks—also number of entries in	24
	count	array_of_displacements (non-negative integer)	25
INI	blacklangth		26
IN	blocklength	number of elements in each block (non-negative	27
		integer)	28
IN	array_of_displacements	array of displacements, in multiples of $oldtype$ (array	29
		of integers)	30
IN	oldtype	old datatype (handle)	31
	noutuno	nom datatuma (bandla)	32
OUT	newtype	new datatype (handle)	33

### C binding

C binding	35	
<pre>int MPI_Type_create_indexed_block(int count, int blocklength,</pre>		
<pre>const int array_of_displacements[], MPI_Datatype oldtype,</pre>	37	
MPI_Datatype *newtype)	38	
<pre>int MPI_Type_create_indexed_block_c(MPI_Count count, MPI_Count blocklength,</pre>	39	
const MPI_Count array_of_displacements[],	40	
MPI_Datatype oldtype, MPI_Datatype *newtype)	41	
MI_Datatype Oldtype, MI_Datatype *newtype)	42	
Fortran 2008 binding		
<pre>MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,</pre>	44	
oldtype, newtype, ierror)	45	
	46	
INTEGER, INTENT(IN) :: count, blocklength,		
INTEGER, INTENT(IN) :: Count, DIOCKIENGTH, array_of_displacements(count)	47	
	47 48	

 $\mathbf{2}$ 

1		(MPI_Datatype), INTENT(OU ER, OPTIONAL, INTENT(OUT	V -		
3 4 5 6 7 8 9 10	<pre>MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,</pre>				
11 12 13 14 15 16	Fortran binding MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR				
17 18 19 20 21	Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.				
22 23 24	MPI_TYPI	E_CREATE_HINDEXED_BLO oldtype, newtype)	CK(count, blocklength, array_of_displacements,		
25 26	IN	count	number of blocks—also number of entries in array_of_displacements (non-negative integer)		
27 28	IN	blocklength	number of elements in each block (non-negative integer)		
29 30	IN	array_of_displacements	byte displacement of each block (array of integers)		
31	IN	oldtype	old datatype (handle)		
32 33	OUT	newtype	new datatype (handle)		
34 35 36 37 38	C binding int MPI_Type_create_hindexed_block(int count, int blocklength, const MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)				
39 40 41 42	int MPI_7				
43 44 45 46 47 48	MPI_Type_	2008 binding create_hindexed_block(cc oldtype, newtype, id ER, INTENT(IN) :: count,			

INTE	EGER(KIND=MPI_ADDRESS_KIN	D), INTENT(IN) ::	1
	array_of_displacem		2
	E(MPI_Datatype), INTENT(I		3
	E(MPI_Datatype), INTENT(O		4
INTE	EGER, OPTIONAL, INTENT(OU	T) :: ierror	5 6
MPI_Type	e_create_hindexed_block(c	ount, blocklength, array_of_displacements,	6 7
01	oldtype, newtype, i		8
INTE	EGER(KIND=MPI_COUNT_KIND)	, INTENT(IN) :: count, blocklength,	9
	array_of_displacem		10
	E(MPI_Datatype), INTENT(I		11
	E(MPI_Datatype), INTENT(O	• -	12
INTE	EGER, OPTIONAL, INTENT(OU	T) :: ierror	13
Fortran	binding		14
MPI_TYPE	E_CREATE_HINDEXED_BLOCK(C	OUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,	15
	OLDTYPE, NEWTYPE, I	ERROR)	16
	EGER COUNT, BLOCKLENGTH,		17
INTE	EGER(KIND=MPI_ADDRESS_KIN	D) ARRAY_OF_DISPLACEMENTS(*)	18
			19 20
Struct N	MPI TYPE CREATE STRUCT	is the most general type constructor. It further	20 21
		EXED in that it allows each block to consist of repli-	22
0	f different datatypes.	*	23
			24
	DE CREATE STRUCT/sound	array of blocklongthe array of displacements	25
	array_of_types, newtyp	array_of_blocklengths, array_of_displacements,	26
		,	27
IN	count	number of blocks—also number of entries in arrays	28
		array_of_types, array_of_displacements, and	29
		array_of_blocklengths (non-negative integer)	30
IN	array_of_blocklengths	number of elements in each block (array of	31
		non-negative integers)	32 33
IN	array_of_displacements	byte displacement of each block (array of integers)	34
IN	array_of_types	type of elements in each block (array of handles)	35
OUT	newtype	new datatype (handle)	36
001	newtype	new datatype (nandle)	37
C bindi	ng		38
	•	<pre>ount, const int array_of_blocklengths[],</pre>	39
1110 III 1 <u>.</u>		<pre>y_of_displacements[],</pre>	40
		array_of_types[], MPI_Datatype *newtype)	41
			42
int MPI_	_Type_create_struct_c(MPI		43
		<pre>ray_of_blocklengths[], ray_of_displacements[]</pre>	44
		ay_of_displacements[], array_of_types[], MPI_Datatype *newtype)	45 46
	const mi_Datatype	array_or_oypes[], mr_bacacype *newcype)	46 47
			48

```
1
            Fortran 2008 binding
 \mathbf{2}
            MPI_Type_create_struct(count, array_of_blocklengths,
 3
                                               array_of_displacements, array_of_types, newtype, ierror)
 4
                       INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
 5
                       INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
 6
                                                 array_of_displacements(count)
 7
                       TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
 8
                       TYPE(MPI_Datatype), INTENT(OUT) :: newtype
 9
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
            MPI_Type_create_struct(count, array_of_blocklengths,
11
                                               array_of_displacements, array_of_types, newtype, ierror) !(_c)
12
                       INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
13
                                                 array_of_blocklengths(count), array_of_displacements(count)
14
                       TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
15
                       TYPE(MPI_Datatype), INTENT(OUT) :: newtype
16
                       INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
            Fortran binding
19
             MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
20
                                               ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
21
                       INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
22
                                                 IERROR
23
                       INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
^{24}
25
             Example 5.6 Let type1 have type map,
26
                          \{(double, 0), (char, 8)\},\
27
28
             with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR).
29
             Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with
30
             type map,
^{31}
32
                          \{(\texttt{float}, 0), (\texttt{float}, 4), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{char}, 26), (\texttt{char}, 27), (\texttt{char}, 28)\}.
33
             That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at
34
             16, followed by three copies of MPI_CHAR, starting at 26. In this example, we assume that
35
             a float occupies four bytes.
36
37
                      In general, let T be the array_of_types argument, where T[i] is a handle to,
38
                         typemap_i = \{(type_0^i, disp_0^i), \dots, (type_{n-1}^i, disp_{n-1}^i)\},\
39
40
             with extent ex_i. Let B be the array_of_blocklength argument and D be the
41
             array_of_displacements argument. Let c be the count argument. Then the newly created
42
            datatype has a type map with \sum_{i=0}^{C-1} B[i] \cdot n_i entries:
43
44
                         \{(type_0^0, disp_0^0 + \mathsf{D}[0]), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0]), \dots, \}
45
                         (type_0^0, disp_0^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] - 1) \cdot ex_0), \dots, (type_{n_0}^0 + \mathsf{D}[0] + (\mathsf{B}[0] + \mathsf{D}[0] + (\mathsf{B}[0] 
46
47
                        (type_0^{\mathsf{C}-1}, disp_0^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1]), \dots, (type_{n_{\mathsf{C}-1}-1}^{\mathsf{C}-1}, disp_{n_{\mathsf{C}-1}-1}^{\mathsf{C}-1} + \mathsf{D}[\mathsf{c}-1]), \dots,
48
```

$(type_0^{C-1}, disp_0^{C-1} + D[c-1] + (B[c-1] - 1) \cdot ex_{C-1}), \dots,$	1
	2
$(type_{n_{C-1}-1}^{C-1}, disp_{n_{C-1}-1}^{C-1} + D[c-1] + (B[c-1]-1) \cdot ex_{C-1})\}.$	3
	4
A call to $MPI_TYPE_CREATE_HINDEXED$ (count, B, D, oldtype, newtype) is equivalent	5
to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of T	6
is equal to oldtype.	7
	8
5.1.3 Subarray Datatype Constructor	9
	10
	11
MDI TYDE CREATE SURAPRAV(ndime array of cizes array of subsizes array of starts	12

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts,			
	order, oldtype, newtype)	,,,,,,,,,,,,,,	13
	P		14
IN	ndims	number of array dimensions (positive integer)	15
IN	array_of_sizes	number of elements of type oldtype in each dimension	16
		of the full array (array of positive integers)	17
IN	array_of_subsizes	number of elements of type oldtype in each dimension	18
IIN		of the subarray (array of positive integers)	19
		of the subarray (array of positive integers)	20
IN	array_of_starts	starting coordinates of the subarray in each	21
		dimension (array of non-negative integers)	

		of the subarray (array of positive integers)
IN	array_of_starts	starting coordinates of the subarray in each dimension (array of non-negative integers)
IN	order	array storage order flag (state)
IN	oldtype	old datatype (handle)
OUT	newtype	new datatype (handle)

# C binding

<pre>int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],</pre>	29
<pre>const int array_of_subsizes[], const int array_of_starts[],</pre>	30
int order, MPI_Datatype oldtype, MPI_Datatype *newtype)	31
<pre>int MPI_Type_create_subarray_c(int ndims, const MPI_Count array_of_sizes[],</pre>	32
const MPI_Count array_of_subsizes[],	33
<pre>const MPI_Count array_of_starts[], int order,</pre>	34
MPI_Datatype oldtype, MPI_Datatype *newtype)	35
In 1_bababype blabype, in 1_bababype ineweype,	36
Fortran 2008 binding	37
<pre>MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,</pre>	38
array_of_starts, order, oldtype, newtype, ierror)	39
<pre>INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),</pre>	40
<pre>array_of_subsizes(ndims), array_of_starts(ndims), order</pre>	41
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	42
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,	45
array_of_starts, order, oldtype, newtype, ierror) !(_c)	46
INTEGER, INTENT(IN) :: ndims, order	47
INTEGER, INTENT(IN) HOIMS, OFGET	48

```
1
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_sizes(ndims),
2
                      array_of_subsizes(ndims), array_of_starts(ndims)
3
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
4
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
     Fortran binding
7
     MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
8
                     ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
9
          INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
10
                      ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
11
12
          The subarray type constructor creates an MPI datatype describing an n-dimensional
13
     subarray of an n-dimensional array. The subarray may be situated anywhere within the
14
      full array, and may be of any nonzero size up to the size of the larger array as long as it
15
      is confined within this array. This type constructor facilitates creating filetypes to access
16
      arrays distributed in blocks among processes to a single file that contains the global array,
17
     see MPI I/O, especially Section 14.1.1.
18
          This type constructor can handle arrays with an arbitrary number of dimensions and
19
      works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note
20
      that a C program may use Fortran order and a Fortran program may use C order.
21
          The ndims parameter specifies the number of dimensions in the full data array and
22
      gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.
23
          The number of elements of type oldtype in each dimension of the n-dimensional ar-
^{24}
     ray and the requested subarray are specified by array_of_sizes and array_of_subsizes, re-
25
     spectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] < 1 or
26
      array_of_subsizes[i] > array_of_sizes[i].
27
          The array_of_starts contains the starting coordinates of each dimension of the subarray.
28
      Arrays are assumed to be indexed starting from zero. For any dimension i, it is erroneous to
29
      specify array_of_starts[i] < 0 or array_of_starts[i] > (array_of_sizes[i] - array_of_subsizes[i]).
30
           Advice to users. In a Fortran program with arrays indexed starting from 1, if the
^{31}
           starting coordinate of a particular dimension of the subarray is n, then the entry in
32
           array_of_starts for that dimension is n-1. (End of advice to users.)
33
34
          The order argument specifies the storage order for the subarray as well as the full array.
35
      It must be set to one of the following:
36
37
      MPI_ORDER_C The ordering used by C arrays, (i.e., row-major order)
38
      MPI_ORDER_FORTRAN The ordering used by Fortran arrays, (i.e., column-major order)
39
40
          A ndims-dimensional subarray (newtype) with no extra padding can be defined by the
^{41}
      function Subarray() as follows:
42
           newtype = Subarray(ndims, {size_0, size_1, \ldots, size_{ndims-1}},
43
                         \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\
44
                         {start_0, start_1, \ldots, start_{ndims-1}}, oldtype)
45
46
          Let the typemap of oldtype have the form:
47
48
           \{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}
```

where  $type_i$  is a predefined MPI datatype, and let ex be the extent of oldtype. Then we define the Subarray() function recursively using the following three equations. Equation 5.2 defines the base step. Equation 5.3 defines the recursion step when order = MPI\_ORDER\_FORTRAN, and Equation 5.4 defines the recursion step when order = MPI\_ORDER\_C. These equations use the conceptual datatypes lb\_marker and ub\_marker; see Section 5.1.6 for details.

		7
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\},\$	(5.2)	8
$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\})$		9
$= \{(lb_marker, 0),$		10
$(type_0, disp_0 + start_0 \times ex), \dots, (type_{n-1}, disp_{n-1} + start_0 \times ex),$		11 12
$(type_0, disp_0 + (start_0 + 1) \times ex), \dots, (type_{n-1},$		12
$disp_{n-1} + (start_0 + 1) \times ex), \dots$		14
$(type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \dots,$		15
		16
$(type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex),$		17
$(ub\_marker, size_0 \times ex) \}$		18
		19
Subarray( $ndims$ , { $size_0, size_1, \ldots, size_{ndims-1}$ },	(5.3)	20
$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$		21
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}$		22 23
$= \text{Subarray}(ndims - 1, \{size_1, size_2, \dots, size_{ndims-1}\},\$		23 24
$\{subsize_1, subsize_2, \dots, subsize_{ndims-1}\},\$		25
		26
$\{start_1, start_2, \dots, start_{ndims-1}\},\$		27
$Subarray(1, \{size_0\}, \{subsize_0\}, \{start_0\}, oldtype))$		28
		29
Subarray $(ndims, \{size_0, size_1, \dots, size_{ndims-1}\},\$	(5.4)	30
$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},\$		31
$\{start_0, start_1, \dots, start_{ndims-1}\}, oldtype\}$		32
= Subarray( $ndims - 1$ , { $size_0, size_1, \ldots, size_{ndims-2}$ },		33
$\{subsize_0, subsize_1, \dots, subsize_{ndims-2}\},\$		34 35
$\{start_0, start_1, \dots, start_{ndims-2}\},\$		36
	1	37
Subarray(1, $\{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, old \}$	ітуре))	38

For an example use of MPI\_TYPE\_CREATE\_SUBARRAY in the context of I/O see Section 14.9.2.

### 5.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [47] data distributions. However, unlike in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as follows. Complementary filetypes are created by having every process of a group call 48

1 2

3 4

5

6 7

39

40

 $41 \\ 42$ 

43

44

 $45 \\ 46$ 

	136		CHAPTER 5. DATATYPES	
1 2 3 4 5 6 7 8 9	this constructor with identical arguments (with the exception of rank which should be set appropriately). These filetypes (along with identical disp and etype) are then used to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 14.1.1 and Section 14.3. Using this view, a collective data access operation (with identical offsets) will yield an HPF-like distribution pattern. ( <i>End of advice to users.</i> ) MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs,			
10		array_of_dargs, array_of_	psizes, order, oldtype, newtype)	
11	IN	size	size of process group (positive integer)	
12	IN	rank	rank in process group (non-negative integer)	
13 14 15	IN	ndims	number of array dimensions as well as process grid dimensions (positive integer)	
16 17	IN	array_of_gsizes	number of elements of type <b>oldtype</b> in each dimension of global array (array of positive integers)	
18 19	IN	array_of_distribs	distribution of array in each dimension (array of states)	
20 21 22	IN	array_of_dargs	distribution argument in each dimension (array of positive integers)	
23 24	IN	array_of_psizes	size of process grid in each dimension (array of positive integers)	
25	IN	order	array storage order flag (state)	
26	IN	oldtype	old datatype (handle)	
27 28	OUT	newtype	new datatype (handle)	
29 30 31 32 33 34	C binding int MPI_Type_create_darray(int size, int rank, int ndims, const int array_of_gsizes[], const int array_of_distribs[], const int array_of_dargs[], const int array_of_psizes[],			
35 36 37 38 39 40	<pre>int MPI_Type_create_darray_c(int size, int rank, int ndims,</pre>			
41 42 43 44 45 46 47 48	<pre>Fortran 2008 binding MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,</pre>			

TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 $2$
<pre>MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,</pre>	3 4 5 6 7 8 9 10 11 12 13
<pre>Fortran binding MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,</pre>	14 15 16 17 18 19 20
MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be set to 1 (see Example 5.7). For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array_of_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.	20 21 22 23 24 25 26 27
Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 8. ( <i>End of advice to users.</i> )	28 29 30 31 32 33
Each dimension of the array can be distributed in one of three ways:	34 35
• MPI_DISTRIBUTE_BLOCK - Block distribution	36
• MPI_DISTRIBUTE_CYCLIC - Cyclic distribution	37 38
$\bullet$ MPI_DISTRIBUTE_NONE - Dimension not distributed	39
The constant MPI_DISTRIBUTE_DFLT_DARG specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension i in which the distribution is MPI_DISTRIBUTE_BLOCK, it is erroneous to specify array_of_dargs[i] * array_of_psizes[i] < array_of_gsizes[i]. For example, the HPF layout ARRAY(CYCLIC(15)) corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout AR- RAY(BLOCK) corresponds to MPI_DISTRIBUTE_BLOCK with a distribution argument of MPI_DISTRIBUTE_DFLT_DARG.	40 41 42 43 44 45 46 47 48

1	The order argument is used as in MPI_TYPE_CREATE_SUBARRAY to specify the stor-
2	age order. Therefore, arrays described by this type constructor may be stored in Fortran
3	(column-major) or C (row-major) order. Valid values for order are MPI_ORDER_FORTRAN
4	and MPI_ORDER_C.
5	
	This routine creates a new MPI datatype with a typemap defined in terms of a function
6	called "cyclic()" (see below).
7	Without loss of generality, it suffices to define the typemap for the
8	MPI_DISTRIBUTE_CYCLIC case where MPI_DISTRIBUTE_DFLT_DARG is not used.
9	$MPI_DISTRIBUTE_BLOCK$ and $MPI_DISTRIBUTE_NONE$ can be reduced to the
10	MPI_DISTRIBUTE_CYCLIC case for dimension i as follows.
11	MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG
12	is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to
13	
14	$(array\_of\_gsizes[i] + array\_of\_psizes[i] - 1) / array\_of\_psizes[i].$
15	If array_of_dargs[i] is not MPI_DISTRIBUTE_DFLT_DARG, then MPI_DISTRIBUTE_BLOCK and
16	MPI_DISTRIBUTE_CYCLIC are equivalent.
17	MPI_DISTRIBUTE_NONE is equivalent to MPI_DISTRIBUTE_CYCLIC with
18	•
19	array_of_dargs[i] set to array_of_gsizes[i].
	Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to
20	MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with
21	array_of_dargs[i] set to 1.
22	For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined
23	by the following code fragment:
24	
25	oldtypes[0] = oldtype;
26	for (i = 0; i < ndims; i++) {
27	<pre>oldtypes[i+1] = cyclic(array_of_dargs[i],</pre>
28	array_of_gsizes[i],
29	r[i],
30	array_of_psizes[i],
31	oldtypes[i]);
32	}
33	<pre>newtype = oldtypes[ndims];</pre>
34	For MPI_ORDER_C, the code is:
35	FOR MET_ONDER_C, the code is.
36	<pre>oldtypes[0] = oldtype;</pre>
37	for (i = 0; i < ndims; i++) {
38	oldtypes[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
39	
40	array_of_gsizes[ndims - i - 1],
41	r[ndims - i - 1],
42	$array_of_psizes[ndims - i - 1],$
	<pre>oldtypes[i]);</pre>
43	}
44	<pre>newtype = oldtypes[ndims];</pre>
45	**
46	
47	where r[i] is the position of the process (with rank rank) in the process grid at dimension
48	i The values of r[i] are given by the following code fragment:

<pre>t_rank = rank; t_size = 1; for (i = 0; i &lt; ndims; i++) t_size *= array_of_psizes[i]; for (i = 0; i &lt; ndims; i++) { t_size = t_size / array_of_psizes[i]; r[i] = t_rank / t_size;</pre>	1 2 3 4 5 6 7
<pre>t_rank = t_rank % t_size; }</pre>	8 9
Let the typemap of <b>oldtype</b> have the form:	10
	11 12
$\{(type_0, disp_0), (type_1, disp_1), \dots, (type_{n-1}, disp_{n-1})\}$	13
where $type_i$ is a predefined MPI datatype, and let $ex$ be the extent of oldtype. The following function uses the conceptual datatypes lb_marker and ub_marker, see Section 5.1.6 for details.	14 15 16
Given the above, the function cyclic() is defined as follows:	17
$\operatorname{cyclic}(darg,gsize,r,psize,oldtype)$	18 19
$= \{(lb\_marker, 0),$	20
$(type_0, disp_0 + r \times darg \times ex), \ldots,$	21
$(type_{n-1}, disp_{n-1} + r \times darg \times ex),$	22
$(type_0, disp_0 + (r \times darg + 1) \times ex), \dots,$	23 24
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex),$	25
	26
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex), \dots,$	27
$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex),$	28 29 30
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,$	31
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex),$	32
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \dots,$	33
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex),$	34 35 36
$(type_0, disp_0 + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex), \dots,$	37
$(type_{n-1}, disp_{n-1} + ((r+1) \times darg - 1) \times ex + psize \times darg \times ex),$ :	38 39
$(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots,$	40 41
$(type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \dots, (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)),$	41
$(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)),$	43
$(type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex$	44
$(sgp \circ_{n-1}, usp_{n-1} + (r \times usr g + 1) \times vsu + psize \times darg \times ex \times (count - 1)),$	45
	$46 \\ 47$
$(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex$	48

```
1
                                +psize \times darg \times ex \times (count - 1)), \ldots,
\mathbf{2}
                        (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex
3
                                +psize \times darq \times ex \times (count - 1)),
4
                 (ub\_marker, gsize * ex)
5
6
      where count is defined by this code fragment:
7
          nblocks = (gsize + (darg - 1)) / darg;
8
          count = nblocks / psize;
9
          left_over = nblocks - count * psize;
10
          if (r < left_over)</pre>
11
               count = count + 1;
12
13
      Here, nblocks is the number of blocks that must be distributed among the processors.
14
      Finally, darg_{last} is defined by this code fragment:
15
          if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
16
               darg_last = darg;
17
          else {
18
               darg_last = num_in_last_cyclic - darg * r;
19
               if (darg_last > darg)
20
                    darg_last = darg;
21
               if (darg_last <= 0)</pre>
22
                   darg_last = darg;
23
          }
^{24}
25
      Example 5.7 Consider generating the filetypes corresponding to the HPF distribution:
26
27
             <oldtype> FILEARRAY(100, 200, 300)
28
      !HPF$ PROCESSORS PROCESSES(2, 3)
29
      !HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
30
      This can be achieved by the following Fortran code, assuming there will be six processes
^{31}
      attached to the run:
32
33
      ndims = 3
34
      array_of_gsizes(1) = 100
35
      array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
36
      array_of_dargs(1) = 10
37
      array_of_gsizes(2) = 200
38
      array_of_distribs(2) = MPI_DISTRIBUTE_NONE
39
      \operatorname{array_of_dargs}(2) = 0
40
      array_of_gsizes(3) = 300
41
      array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
42
      array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
43
      array_of_psizes(1) = 2
44
      array_of_psizes(2) = 1
45
      array_of_psizes(3) = 3
46
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
47
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
48
```

#### 5.1. DERIVED DATATYPES

```
call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
    array_of_distribs, array_of_dargs, array_of_psizes, &
    MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
```

## 5.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI\_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI\_BOTTOM. Note that in Fortran MPI\_BOTTOM is not usable for initialization or assignment, see Section 2.5.4.

The address of a location in memory can be found by invoking the function MPI\_GET\_ADDRESS. The **relative displacement** between two absolute addresses can be calculated with the function MPI\_AINT\_DIFF. A new absolute address as sum of an absolute base address and a relative displacement can be calculated with the function MPI\_AINT\_ADD. To ensure portability, arithmetic on absolute addresses should not be performed with the intrinsic operators "-" and "+". See also Sections 2.5.6 and 5.1.12 on pages 22 and 156.

Rationale. Address sized integer values, i.e., MPI\_Aint or INTEGER(KIND=MPI\_ADDRESS\_KIND) values, are signed integers, while absolute addresses are unsigned quantities. Direct arithmetic on addresses stored in address sized signed variables can cause overflows, resulting in undefined behavior. (End of rationale.)

MPI_GET_	ADDRESS(location, address)	
IN	location	location in caller memory (choice)
OUT	address	address of location (integer)

## C binding

int MPI\_Get\_address(const void \*location, MPI\_Aint \*address)

```
Fortran 2008 binding
```

MPI\_Get\_address(location, address, ierror)
 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: location
 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: address
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

Returns the (byte) address of location.

 $\mathbf{2}$ 

1 2 3 4	IN	TENT(IN) becaus	he mpi_f08 module, the location argument is not defined with se existing applications may use MPI_GET_ADDRESS as a substi- NC_REG, which was not defined before MPI-3.0. ( <i>End of rationale.</i> )	
5 6	Examp	le 5.8 Using M	PI_GET_ADDRESS for an array.	
7 8 9 10 11 12 13 14	INTEGEN CALL MH CALL MH DIFF = ! The v	PI_GET_ADDRESS PI_GET_ADDRESS MPI_AINT_DIFF	is 909*SIZEOF(REAL); the values of I1 and I2 are	
15 16 17 18 19 20 21 22 23	Mi ho th th us	wever, that & cas at the value of a e object pointed ay not have a un e of MPI_GET_A	C users may be tempted to avoid the usage of S and rely on the availability of the address operator &. Note, <i>st-expression</i> is a pointer, not an address. ISO C does not require a pointer (or the pointer cast to int) be the absolute address of at—although this is commonly the case. Furthermore, referencing ique definition on machines with a segmented address space. The DDRESS to "reference" C variables guarantees portability to such End of advice to users.)	
24 25 26 27	op	lvice to users. timization done .1.20. (End of ac	To prevent problems with the argument copying and register by Fortran compilers, please note the hints in Sections 19.1.10– lvice to users.)	
28 29 30 31		-	y, arithmetic on MPI addresses must be performed using the PI_AINT_DIFF functions.	
32	MPI_AII	NT_ADD(base, di	sp)	
33 34	IN	base	base address (integer)	
35	IN	disp	displacement (integer)	
36 37 38		C binding MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)		
39 40 41 42	INTEGER	Fortran 2008 binding INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp) INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp		
43 44 45	INTEGER	Fortran binding INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP) INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP		
46 47 48		MPI_AINT_ADD produces a new MPI_Aint value that is equivalent to the sum of the base and disp arguments, where base represents a base address returned by a call to		

MPI\_GET\_ADDRESS and disp represents a signed integer displacement. The resulting address is valid only at the process that generated base, and it must correspond to a location in the same object referenced by base, as described in Section 5.1.12. The addition is performed in a manner that results in the correct MPI\_Aint representation of the output address, as if the process that originally produced base had called:

```
MPI_Get_address((char *) base + disp, &result);
```

MPI_AINT_DIFF(addr1, addr2)				
IN	addr1	minuend address (integer)		
IN	addr2	subtrahend address (integer)		

C binding

MPI\_Aint MPI\_Aint\_diff(MPI\_Aint addr1, MPI\_Aint addr2)
Fortran 2008 binding
INTEGER(KIND=MPI\_ADDRESS\_KIND) MPI\_Aint\_diff(addr1, addr2)

INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: addr1, addr2

Fortran binding

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2)
INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2
```

MPI\_AINT\_DIFF produces a new MPI\_Aint value that is equivalent to the difference between addr1 and addr2 arguments, where addr1 and addr2 represent addresses returned by calls to MPI\_GET\_ADDRESS. The resulting address is valid only at the process that generated addr1 and addr2, and addr1 and addr2 must correspond to locations in the same object in the same process, as described in Section 5.1.12. The difference is calculated in a manner that results in the signed difference from addr1 to addr2, as if the process that originally produced the addresses had called (char \*) addr1 - (char \*) addr2 on the addresses initially passed to MPI\_GET\_ADDRESS.

The following auxiliary functions provide useful information on derived datatypes.

MPI\_TYPE\_SIZE(datatype, size)

IN	datatype	datatype to get information on (handle)	00
	adatype	datatype to get information on (nandle)	37
OUT	size	datatype size (integer)	38
			39
C bindin	g		40
int MPI_7	Type_size(MPI_Datatype dat	catype, int *size)	41
int MDT "	int MDI Turne sine s(MDI Deteture deteture MDI Count vision)		
IIIC MFI	<pre>int MPI_Type_size_c(MPI_Datatype datatype, MPI_Count *size)</pre>		
Fortran 2	Fortran 2008 binding		
MPI_Type	MPI_Type_size(datatype, size, ierror)		
TYPE	TYPE(MPI_Datatype), INTENT(IN) :: datatype		
INTE	GER, INTENT(OUT) :: size		47
INTEC	GER, OPTIONAL, INTENT(OUT)	:: ierror	48

```
1
     MPI_Type_size(datatype, size, ierror) !(_c)
\mathbf{2}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     Fortran binding
6
     MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
7
         INTEGER DATATYPE, SIZE, IERROR
8
9
10
     MPI_TYPE_SIZE_X(datatype, size)
11
12
       IN
                datatype
                                           datatype to get information on (handle)
13
       OUT
                size
                                           datatype size (integer)
14
15
     C binding
16
     int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
17
18
     Fortran 2008 binding
19
     MPI_Type_size_x(datatype, size, ierror)
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     Fortran binding
^{24}
     MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
25
         INTEGER DATATYPE, IERROR
26
         INTEGER(KIND=MPI_COUNT_KIND) SIZE
27
28
         MPI_TYPE_SIZE and MPI_TYPE_SIZE_X set the value of size to the total size, in
29
```

<sup>29</sup> bytes, of the entries in the type signature associated with datatype; i.e., the total size of the <sup>30</sup> data in a message that would be created with this datatype. Entries that occur multiple <sup>31</sup> times in the datatype are counted with their multiplicity. For both functions, if the OUT <sup>32</sup> parameter cannot express the value to be returned (e.g., if the parameter is too small to <sup>33</sup> hold the output value), it is set to MPI\_UNDEFINED.

35 36

## 5.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, 37 and override the definition given on page 145. This allows one to define a datatype that has 38"holes" at its beginning or its end, or a datatype with entries that extend above the upper 39 bound or below the lower bound. Examples of such usage are provided in Section 5.1.14. 4041 Also, the user may want to overide the alignment rules that are used to compute upper 42bounds and extents. E.g., a C compiler may allow the user to overide default alignment rules for some of the structures within a program. The user has to specify explicitly the 43bounds of the datatypes that match these structures. 44

To achieve this, we add two additional conceptual datatypes, **lb\_marker** and **ub\_marker**, that represent the lower bound and upper bound of a datatype. These conceptual datatypes occupy no space (*extent*(**lb\_marker**) = *extent*(**ub\_marker**) = 0). They do not affect the size or count of a datatype, and do not affect the content of a message created In general, if

with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

**Example 5.9** A call to MPI\_TYPE\_CREATE\_RESIZED(MPI\_INT, -3, 9, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the typemap {(lb\_marker, -3), (int, 0), (ub\_marker, 6)}. If this type is replicated twice by a call to MPI\_TYPE\_CONTIGUOUS(2, type1, type2) then the newly created type can be described by the typemap {(lb\_marker, -3), (int, 0), (int, 0), (int, 9), (ub\_marker, 15)}. (An entry of type ub\_marker can be deleted if there is another entry of type ub\_marker with a higher displacement; an entry of type lb\_marker can be deleted if there is another entry of type lb\_marker with a lower displacement.)

 $Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\},\$ then the **lower bound** of *Typemap* is defined to be  $lb(Typemap) = \begin{cases} \min_{j} disp_{j} & \text{Ib marker} \\ \min_{j} \{ disp_{j} \text{ such that } type_{j} = \text{Ib}_{marker} \} & \text{otherwise} \end{cases}$ if no entry has type Similarly, the **upper bound** of *Typemap* is defined to be  $ub(Typemap) = \begin{cases} \max_{j}(disp_{j} + sizeof(type_{j})) + \epsilon & \text{if no entry} \\ \max_{j}\{disp_{j} \text{ such that } type_{j} = ub\_marker\} & \text{otherwise} \end{cases}$ if no entry has type ub\_marker Then extent(Typemap) = ub(Typemap) - lb(Typemap)If  $type_i$  requires alignment to a byte address that is a multiple of  $k_i$ , then  $\epsilon$  is the least non-negative increment needed to round extent(Typemap) to the next multiple of  $\max_i k_i$ . In Fortran, it is implementation dependent whether the MPI implementation computes the alignments  $k_i$  according to the alignments used by the compiler in common blocks, SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE nor BIND(C). The formal definitions given for the various datatype constructors apply now, with the amended definition of extent. *Rationale.* Before Fortran 2003, MPI\_TYPE\_CREATE\_STRUCT could be applied to

Fortran common blocks and SEQUENCE derived types. With Fortran 2003, this list was extended by BIND(C) derived types and MPI implementors have implemented the alignments  $k_i$  differently, i.e., some based on the alignments used in SEQUENCE derived types, and others according to BIND(C) derived types. (*End of rationale.*)

Advice to implementors. In Fortran, it is generally recommended to use BIND(C) derived types instead of common blocks or SEQUENCE derived types. Therefore it is recommended to calculate the alignments  $k_i$  based on BIND(C) derived types. (End of advice to implementors.)

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{31}$ 

Advice to users. Structures combining different basic datatypes should be defined so that there will be no gaps based on alignment rules. If such a datatype is used to create an array of structures, users should also avoid an alignment-gap at the end of the structure. In MPI communication, the content of such gaps would not be communicated into the receiver's buffer. For example, such an alignment-gap may occur between an odd number of floats or REALs before a double or DOUBLE PRECISION data. Such gaps may be added explicitly to both the structure and the MPI derived datatype handle because the communication of a contiguous derived datatype may be significantly faster than the communication of one that is noncontiguous because of such alignment-gaps.

As an example, instead of

TYPE, BIND(C) :: my\_data REAL, DIMENSION(3) :: x ! there may be a gap of the size of one REAL ! if the alignment of a DOUBLE PRECISION is ! two times the size of a REAL DOUBLE PRECISION :: p END TYPE

one should define

```
TYPE, BIND(C) :: my_data
REAL, DIMENSION(3) :: x
REAL :: gap1
DOUBLE PRECISION :: p
END TYPE
```

and also include gap1 in the matching MPI derived datatype. It is required that all processes in a communication add the same gaps, i.e., defined with the same basic datatype. Both the original and the modified structures are portable, but may have different performance implications for the communication and memory accesses during computation on systems with different alignment values.

In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a  $max_ik_i$  alignment less than this structure alignment. To maintain portability, users should always resize structure derived datatype handles if used in an array of structures, see the Example in Section 19.1.15. (*End of advice to users.*)

- 40 41
- 42
- 43
- 44 45
- 46
- 47 48

1

2

3

4

5

6

7

8

9

10

11

12 13

14

15

16

17

18

19

20 21

22 23 24

25

26

27

28 29 30

 $^{31}$ 

32

33

34

35

36

37

38

# 5.1.7 Extent and Bounds of Datatypes

# MPI\_TYPE\_GET\_EXTENT(datatype, lb, extent)

			5		
IN	datatype	datatype to get information on (handle)	6		
OUT	lb	lower bound of datatype (integer)	7		
OUT	extent	extent of datatype (integer)	8		
			9		
C bindi	ıg		10 11		
int MPI_	Type_get_extent(MF	PI_Datatype datatype, MPI_Aint *lb,	11		
	MPI_Aint *ex	tent)	13		
int MPI	Type get extent c(	(MPI_Datatype datatype, MPI_Count *1b,	14		
-	MPI_Count *e		15		
Fortron	2008 binding		16		
	2008 binding	vpe, lb, extent, ierror)	17		
		ITENT(IN) :: datatype	18		
		ESS_KIND), INTENT(OUT) :: 1b, extent	19		
		TENT(OUT) :: ierror	20		
			21 22		
		<pre>rpe, lb, extent, ierror) !(_c)</pre>	22		
	• -	NTENT(IN) :: datatype	23		
		T_KIND), INTENT(OUT) :: lb, extent TENT(OUT) :: ierror	25		
	GER, OFIIONAL, INI		26		
Fortran			27		
		YPE, LB, EXTENT, IERROR)	28		
	CGER DATATYPE, IERF		29		
INTE	GER(KIND=MPI_ADDRE	ESS_KIND) LB, EXTENT	30		
			31		
			32		
MPI_IYF	PE_GET_EXTENT_X(	datatype, Ib, extent)	33		
IN	datatype	datatype to get information on (handle)	34		
OUT	lb	lower bound of datatype (integer)	35 36		
OUT	extent	extent of datatype (integer)	30		
			38		
C bindi	ነው		39		
	0	(MPI_Datatype datatype, MPI_Count *1b,	40		
-	MPI_Count *e		41		
<b>T</b> (			42		
Fortran 2008 binding MPI_Type_get_extent_x(datatype, lb, extent, ierror)					
	-	• •	44		
	• •	NTENT(IN) :: datatype F_KIND), INTENT(OUT) :: lb, extent	45		
	INTEGER OPTIONAL INTENT(OUT) · · jerror				
T 14 T T	10110, OI I UIML, 1111		47		
			48		

```
148
```

```
1
     Fortran binding
\mathbf{2}
     MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
3
          INTEGER DATATYPE, IERROR
4
          INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT
5
         Returns the lower bound and the extent of datatype (as defined in Equation 5.1).
6
         For both functions, if either OUT parameter cannot express the value to be returned
7
     (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.
8
          MPI allows one to change the extent of a datatype, using lower bound and upper bound
9
     markers. This provides control over the stride of successive datatypes that are replicated
10
     by datatype constructors, or are replicated by the count argument in a send or receive call.
11
12
13
     MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)
14
       IN
                 oldtype
                                            input datatype (handle)
15
16
       IN
                 lb
                                            new lower bound of datatype (integer)
17
       IN
                 extent
                                            new extent of datatype (integer)
18
       OUT
                                            output datatype (handle)
                 newtype
19
20
     C binding
21
     int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
22
                    MPI_Aint extent, MPI_Datatype *newtype)
23
24
     int MPI_Type_create_resized_c(MPI_Datatype oldtype, MPI_Count lb,
25
                    MPI_Count extent, MPI_Datatype *newtype)
26
     Fortran 2008 binding
27
     MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
28
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
29
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
30
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
31
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) !(_c)
34
          TYPE(MPI_Datatype), INTENT(IN) :: oldtype
35
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: lb, extent
36
          TYPE(MPI_Datatype), INTENT(OUT) :: newtype
37
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     Fortran binding
39
     MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
40
          INTEGER OLDTYPE, NEWTYPE, IERROR
41
42
          INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
43
         Returns in newtype a handle to a new datatype that is identical to oldtype, except that
44
     the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb
45
     + extent. Any previous lb and ub markers are erased, and a new pair of lower bound and
46
     upper bound markers are put in the positions indicated by the lb and extent arguments.
47
```

This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

## 5.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 6.5) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent, for example by using MPI\_TYPE\_CREATE\_RESIZED. The functions MPI\_TYPE\_GET\_TRUE\_EXTENT and MPI\_TYPE\_GET\_TRUE\_EXTENT\_X are provided which return the true extent of the datatype.

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)			
IN	datatype datatype to get information on (handle)		
OUT	true_lb	true lower bound of datatype (integer)	
OUT	true_extent	true extent of datatype (integer)	

## C binding

## Fortran 2008 binding

- MPI\_Type\_get\_true\_extent(datatype, true\_lb, true\_extent, ierror)
   TYPE(MPI\_Datatype), INTENT(IN) :: datatype
   INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: true\_lb, true\_extent
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
- MPI\_Type\_get\_true\_extent(datatype, true\_lb, true\_extent, ierror) !(\_c)
   TYPE(MPI\_Datatype), INTENT(IN) :: datatype
   INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: true\_lb, true\_extent
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Fortran binding

MPI\_TYPE\_GET\_TRUE\_EXTENT(DATATYPE, TRUE\_LB, TRUE\_EXTENT, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND=MPI\_ADDRESS\_KIND) TRUE\_LB, TRUE\_EXTENT

```
1
      MPI_TYPE_GET_TRUE_EXTENT_X(datatype, true_lb, true_extent)
2
       IN
                 datatype
                                               datatype to get information on (handle)
3
       OUT
                 true_lb
                                               true lower bound of datatype (integer)
4
5
       OUT
                 true_extent
                                               true extent of datatype (integer)
6
\overline{7}
      C binding
8
      int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
9
                     MPI_Count *true_extent)
10
      Fortran 2008 binding
11
      MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
12
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
      MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
18
          INTEGER DATATYPE, IERROR
19
          INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
20
          true_lb returns the offset of the lowest unit of store which is addressed by the datatype,
21
      i.e., the lower bound of the corresponding typemap, ignoring explicit lower bound mark-
22
      ers. true_extent returns the true size of the datatype, i.e., the extent of the correspond-
23
      ing typemap, ignoring explicit lower bound and upper bound markers, and performing no
^{24}
      rounding for alignment. If the typemap associated with datatype is
25
26
           Typemap = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}
27
28
      Then
29
           true_{lb}(Typemap) = min_{i} \{ disp_{i} : type_{i} \neq lb_{marker}, ub_{marker} \},\
30
^{31}
           true\_ub(Typemap) = max_i \{ disp_i + sizeof(type_i) : type_i \neq lb\_marker, ub\_marker \},\
32
33
      and
34
35
           true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).
36
      (Readers should compare this with the definitions in Section 5.1.6 and Section 5.1.7, which
37
      describe the function MPI_TYPE_GET_EXTENT.)
38
39
          The true_extent is the minimum number of bytes of memory necessary to hold a
      datatype, uncompressed.
40
          For both functions, if either OUT parameter cannot express the value to be returned
41
42
      (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.
43
44
      5.1.9 Commit and Free
45
      A datatype object has to be committed before it can be used in a communication. As
46
      an argument in datatype constructors, uncommitted and also committed datatypes can be
47
      used. There is no need to commit basic datatypes. They are "pre-committed."
48
```

MPI_TYPE_COMMIT(datatype)		1
INOUT datatype	datatype that is committed (handle)	2 3
		4
C binding		5
<pre>int MPI_Type_commit(MPI_Datatype</pre>	*datatype)	6
Fortran 2008 binding		7
<pre>MPI_Type_commit(datatype, ierror)</pre>	)	8
TYPE(MPI_Datatype), INTENT(IN		9
INTEGER, OPTIONAL, INTENT(OUT	Γ) :: ierror	10 11
Fortran binding		12
MPI_TYPE_COMMIT(DATATYPE, IERROR)	)	13
INTEGER DATATYPE, IERROR		14
The commit operation commits the	e datatype, that is, the formal description of a com-	15
_	hat buffer. Thus, after a datatype has been commit-	16
	municate the changing content of a buffer or, indeed,	17 18
the content of different buffers, with different buffers.	fferent starting addresses.	19
Advice to implementors The	system may "compile" at commit time an internal	20
-	that facilitates communication, e.g., change from a	21
	t representation of the datatype, and select the most	22
convenient transfer mechanism. (	End of advice to implementors.)	23
		24
to a no-op.	a committed datatype; in this case, it is equivalent	25 26
		20
<b>Example 5.10</b> The following code frag	gment gives examples of using MPI_TYPE_COMMIT.	28
INTEGER type1, type2		29
CALL MPI_TYPE_CONTIGUOUS(5, MPI_)	• •	30
! new type object		31 32
CALL MPI_TYPE_COMMIT(type1, ierr)		33
! now type1 can be type2 = type1	used for communication	34
	d for communication	35
	to same object as type1)	36
CALL MPI_TYPE_VECTOR(3, 5, 4, MP		37
	type object created	38
CALL MPI_TYPE_COMMIT(type1, ierr)		$\frac{39}{40}$
! now type1 can be	used anew for communication	40 41
		42
		43
MPI_TYPE_FREE(datatype)		44
INOUT datatype	datatype that is freed (handle)	45

C binding

int MPI\_Type\_free(MPI\_Datatype \*datatype)

46

47

1	Fortran 2008 binding			
2	MPI_Type_free(datatype, ierror)			
3	TYPE(MPI_Datatype), INTENT(INOUT) :: datatype			
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
5				
6	Fortran binding			
7	PI_TYPE_FREE(DATATYPE, IERROR)			
8	INTEGER DATATYPE, IERROR			
9	Marks the deteture object accessized with deteture for deallegation and sets det	atura		
10	Marks the datatype object associated with datatype for deallocation and sets datatype			
11	o MPI_DATATYPE_NULL. Any communication that is currently using this datatype			
12	omplete normally. Freeing a datatype does not affect any other datatype that was			
13	rom the freed datatype. The system behaves as if input datatype arguments to de	rived		
13	latatype constructors are passed by value.			
		, <b>.</b>		
15	Advice to implementors. The implementation may keep a reference count of a			
16	communications that use the datatype, in order to decide when to free it. Also	,		
17	may implement constructors of derived datatypes so that they keep pointers to			
18	datatype arguments, rather than copying them. In this case, one needs to keep			
19	of active datatype definition references in order to know when a datatype object	t can		
20	be freed. (End of advice to implementors.)			
21				
22	5.1.10 Duplicating a Datatype			
23				
24				
25	/IPI_TYPE_DUP(oldtype, newtype)			
26	IN oldtype datatype (handle)			
27				
28	OUT     newtype     copy of oldtype (handle)			
29 30				
31	C binding			
32	nt MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)			
33	Fortran 2008 binding			
34	PI_Type_dup(oldtype, newtype, ierror)			
	TYPE(MPI_Datatype), INTENT(IN) :: oldtype			
35	TYPE(MPI_Datatype), INTENT(OUT) :: newtype			
36	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
37	INTEGER, OFFICIAL, INTENT(COT) TETTOT			
38	Fortran binding			
39	PI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)			
40	INTEGER OLDTYPE, NEWTYPE, IERROR			
41	MDI TVDE DUD is a tama construction which doubt of the state of the st	h - ·		
42	MPI_TYPE_DUP is a type constructor which duplicates the existing oldtype wit			
43	ociated key values. For each key value, the respective copy callback function determ			
44	he attribute value associated with this key in the new communicator; one particular a			
45	hat a copy callback may take is to delete the attribute from the new datatype. Ret			

in newtype a new datatype with exactly the same properties as oldtype and any copied cached information, see Section 7.7.4. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 5.1.13. The newtype has the same committed state as the old oldtype.

## 5.1.11 Use of General Datatypes in Communication

Handles to derived datatypes can be passed to a communication call wherever a datatype argument is required. A call of the form MPI\_SEND(buf, count, datatype, ...), where count > 1, is interpreted as if the call was passed a new datatype which is the concatenation of count copies of datatype. Thus, MPI\_SEND(buf, count, datatype, dest, tag, comm) is equivalent to,

```
MPI_TYPE_CONTIGUOUS(count, datatype, newtype)
MPI_TYPE_COMMIT(newtype)
MPI_SEND(buf, 1, newtype, dest, tag, comm)
MPI_TYPE_FREE(newtype).
```

Similar statements apply to all other communication functions that have a **count** and **datatype** argument.

Suppose that a send operation MPI\_SEND(buf, count, datatype, dest, tag, comm) is executed, where datatype has type map,

```
\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\
```

and extent *extent*. (Explicit lower bound and upper bound markers are not listed in the type map, but they affect the value of *extent*.) The send operation sends  $n \cdot \text{count}$  entries, where entry  $i \cdot n + j$  is at location  $addr_{i,j} = \text{buf} + extent \cdot i + disp_j$  and has type  $type_j$ , for  $i = 0, \ldots, \text{count} - 1$  and  $j = 0, \ldots, n-1$ . These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address  $addr_{i,j}$  in the calling program should be of a type that matches  $type_j$ , where type matching is defined as in Section 3.3.1. The message sent contains  $n \cdot \text{count}$  entries, where entry  $i \cdot n + j$  has type  $type_j$ .

Similarly, suppose that a receive operation MPI\_RECV(buf, count, datatype, source, tag, comm, status) is executed, where datatype has type map,

 $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\},\$ 

with extent *extent*. (Again, explicit lower bound and upper bound markers are not listed in the type map, but they affect the value of *extent*.) This receive operation receives  $n \cdot \text{count}$ entries, where entry  $i \cdot n + j$  is at location  $\text{buf} + extent \cdot i + disp_j$  and has type  $type_j$ . If the incoming message consists of k elements, then we must have  $k \leq n \cdot \text{count}$ ; the  $i \cdot n + j$ -th element of the message should have a type that matches  $type_j$ .

**Type matching** is defined according to the type signature of the corresponding datatypes, that is, the sequence of basic type components. Type matching does not depend on some aspects of the datatype definition, such as the displacements (layout in memory) or the intermediate types used.

**Example 5.11** This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
...
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS(4, MPI_REAL, type4, ...)
```

 $^{24}$ 

 $\frac{44}{45}$ 

```
1
     CALL MPI_TYPE_CONTIGUOUS(2, type2, type22, ...)
\mathbf{2}
      . . .
3
      CALL MPI_SEND(a, 4, MPI_REAL, ...)
4
     CALL MPI_SEND(a, 2, type2, ...)
5
     CALL MPI_SEND(a, 1, type22, ...)
6
     CALL MPI_SEND(a, 1, type4, ...)
7
      . . .
8
     CALL MPI_RECV(a, 4, MPI_REAL, ...)
9
     CALL MPI_RECV(a, 2, type2, ...)
10
     CALL MPI_RECV(a, 1, type22, ...)
11
     CALL MPI_RECV(a, 1, type4, ...)
12
     Each of the sends matches any of the receives.
13
14
          A datatype may specify overlapping entries. The use of such a datatype in any com-
15
     munication in association with a buffer updated by the operation is erroneous. (This is
16
     erroneous even if the actual message received is short enough not to write any entry more
17
     than once.)
18
          Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed,
19
     where datatype has type map,
20
           \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}.
21
22
     The received message need not fill all the receive buffer, nor does it need to fill a number of
23
     locations which is a multiple of n. Any number, k, of basic elements can be received, where
24
     0 \le k \le \text{count} \cdot n. The number of basic elements received can be retrieved from status using
25
     the query functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.
26
27
28
     MPI_GET_ELEMENTS(status, datatype, count)
29
                 status
       IN
                                              return status of receive operation (status)
30
^{31}
       IN
                 datatype
                                              datatype used by receive operation (handle)
32
       OUT
                 count
                                              number of received basic elements (integer)
33
34
     C binding
35
     int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,
36
                     int *count)
37
38
     int MPI_Get_elements_c(const MPI_Status *status, MPI_Datatype datatype,
39
                     MPI_Count *count)
40
     Fortran 2008 binding
41
     MPI_Get_elements(status, datatype, count, ierror)
42
          TYPE(MPI_Status), INTENT(IN) :: status
43
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
          INTEGER, INTENT(OUT) :: count
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Get_elements(status, datatype, count, ierror) !(_c)
48
          TYPE(MPI_Status), INTENT(IN) :: status
```

TYPE(MPI_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
Fortran binding MPI_GET_ELEMENTS(STATUS, DATATYPI INTEGER STATUS(MPI_STATUS_SI		4 5 6 7		
MPI_GET_ELEMENTS_X(status, dataty	vpe. count)	8 9 10		
IN status	return status of receive operation (status)	11		
IN datatype	datatype used by receive operation (status)	12		
	*- * - ( <i>)</i>	13		
OUT count	number of received basic elements (integer)	14		
C binding		15 16		
0	_Status *status, MPI_Datatype datatype,	17 18		
		19		
Fortran 2008 binding MPI_Get_elements_x(status, dataty	vne count jerror)	20		
TYPE(MPI_Status), INTENT(IN)	-	21		
TYPE(MPI_Datatype), INTENT(I		22		
INTEGER(KIND=MPI_COUNT_KIND)	, INTENT(OUT) :: count	23 24		
INTEGER, OPTIONAL, INTENT(OU	I) :: ierror	25		
Fortran binding				
MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)				
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR				
INTEGER(KIND=MPI_COUNT_KIND) COUNT				
The datatype argument should match the argument provided by the receive call that				
	ons, if the OUT parameter cannot express the value	31 32		
	is too small to hold the output value), it is set to	33		
MPI_UNDEFINED.	DI CET COUNT (Castian 2.2.5) has a different ha	34		
	PI_GET_COUNT (Section 3.2.5), has a different be- evel entries" received, i.e. the number of "copies" of	35		
	e, MPI_GET_COUNT may return any integer value	36		
	COUNT returns $k$ , then the number of basic elements	37		
	PI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) is	38 39		
	received is not a multiple of $n$ , that is, if the receive	40		
	umber of datatype "copies," then MPI_GET_COUNT	41		
sets the value of count to MPI_UNDEFINED.				
<b>Example 5.12</b> Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.				
		44 45		
CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)				
CALL MPI_TYPE_COMMIT(Type2, ierr)				
•••	···· 4			

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_SEND(a, 2, MPI_REAL, 1, 0, comm, ierr)
   CALL MPI_SEND(a, 3, MPI_REAL, 1, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
   CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
   CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                 ! returns i=1
   CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                 ! returns i=2
   CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
   CALL MPI_GET_COUNT(stat, Type2, i, ierr)
                                                 ! returns i=MPI_UNDEFINED
   CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
                                                ! returns i=3
END IF
```

The functions MPI\_GET\_ELEMENTS and MPI\_GET\_ELEMENTS\_X can also be used after a probe to find the number of elements in the probed message. Note that the MPI\_GET\_COUNT, MPI\_GET\_ELEMENTS, and MPI\_GET\_ELEMENTS\_X return the same values when they are used with basic datatypes as long as the limits of their respective count arguments are not exceeded.

*Rationale.* The extension given to the definition of MPI\_GET\_COUNT seems natural: one would expect this function to return the value of the count argument, when the receive buffer is filled. Sometimes datatype represents a basic unit of data one wants to transfer, for example, a record in an array of records (structures). One should be able to find out how many components were received without bothering to divide by the number of elements in each component. However, on other occasions, datatype is used to define a complex layout of data in the receiver memory, and does not represent a basic unit of data for transfers. In such cases, one needs to use the function MPI\_GET\_ELEMENTS or MPI\_GET\_ELEMENTS\_X. (*End of rationale.*)

Advice to implementors. The definition implies that a receive cannot change the value of storage outside the entries defined to compose the communication buffer. In particular, the definition implies that padding space in a structure should not be modified when such a structure is copied from one process to another. This would prevent the obvious optimization of copying the structure, together with the padding, as one contiguous block. The implementation is free to do this optimization when it does not impact the outcome of the computation. The user can "force" this optimization by explicitly including padding as part of the message. (*End of advice to implementors.*)

36 37 38

39

48

## 5.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous
 locations. Thus, care must be exercised that displacements do not cross from one variable
 to another. Also, in machines with a segmented address space, addresses are not unique
 and address arithmetic has some peculiar properties. Thus, the use of addresses, that is,
 displacements relative to the start address MPI\_BOTTOM, has to be restricted.

<sup>45</sup> Variables belong to the same sequential storage if they belong to the same array,
 <sup>46</sup> to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are
 <sup>47</sup> defined recursively as follows:

1

2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18 19

20

21

22

23

24

25

26

27

28 29

30

31

32

33

34

- 1. The function MPI\_GET\_ADDRESS returns a valid address, when passed as argument a variable of the calling program.
- 2. The **buf** argument of a communication function evaluates to a valid address, when passed as argument a variable of the calling program.
- 3. If v is a valid address, and i is an integer, then v+i is a valid address, provided v and v+i are in the same sequential storage.

A correct program uses only valid addresses to identify the locations of entries in communication buffers. Furthermore, if u and v are two valid addresses, then the (integer) difference u - v can be computed only if both u and v are in the same sequential storage. No other arithmetic operations can be meaningfully Aexecuted on addresses.

The rules above impose no constraints on the use of derived datatypes, as long as they are used to define a communication buffer that is wholly contained within the same sequential storage. However, the construction of a communication buffer that contains variables that are not within the same sequential storage must obey certain restrictions. Basically, a communication buffer with variables that are not within the same sequential storage can be used only by specifying in the communication call buf = MPI\_BOTTOM, count = 1, and using a datatype argument where all displacements are valid (absolute) addresses.

Advice to users. It is not expected that MPI implementations will be able to detect erroneous, "out of bound" displacements—unless those overflow the user address space—since the MPI call may not know the extent of the arrays and records in the host program. (*End of advice to users.*)

Advice to implementors. There is no need to distinguish (absolute) addresses and (relative) displacements on a machine with contiguous address space: MPI\_BOTTOM is zero, and both addresses and displacements are integers. On machines where the distinction is required, addresses are recognized as expressions that involve MPI\_BOTTOM. (*End of advice to implementors.*)

## 5.1.13 Decoding a Datatype

MPI datatype objects allow users to specify an arbitrary layout of data in memory. There are several cases where accessing the layout information in opaque datatype objects would be useful. The opaque datatype object has found a number of uses outside MPI. Furthermore, a number of tools wish to display internal information about a datatype. To achieve this, datatype decoding functions are provided. The two functions in this section are used together to decode datatypes to recreate the calling sequence used in their initial definition. These can be used to allow a user to determine the type map and type signature of a datatype.

 $\overline{7}$ 

 $^{24}$ 

1 2	MPI_TYP	E_GET_ENVELOPE(data num_datatypes, cor	type, num_integers, num_addresses, num_large_counts, nbiner)
3	IN	datatype	datatype to decode (handle)
4 5 6	OUT	num_integers	number of input integers used in call constructing combiner (non-negative integer)
7 8	OUT	num_addresses	number of input addresses used in call constructing <b>combiner</b> (non-negative integer)
9 10 11 12	OUT	num_large_counts	number of input large counts used in call constructing combiner (non-negative integer, only present for large count variants)
12 13 14	OUT	num_datatypes	number of input datatypes used in call constructing <b>combiner</b> (non-negative integer)
15 16	OUT	combiner	combiner (state)
17 18 19 20		Type_get_envelope(MPI int *num_address	_Datatype datatype, int *num_integers, ses, int *num_datatypes, int *combiner)
21 22 23	int MPI_	MPI_Count *num_a	<pre>PI_Datatype datatype, MPI_Count *num_integers, addresses, MPI_Count *num_large_counts, latatypes, int *combiner)</pre>
24 25 26 27 28 29 30 31	MPI_Type TYPE INTE	combiner, ierror (MPI_Datatype), INTEN	T(IN) :: datatype um_integers, num_addresses, num_datatypes,
32 33 34 35 36 37 38	TYPE INTE INTE	num_large_counts (MPI_Datatype), INTEN GER(KIND=MPI_COUNT_KI	ND), INTENT(OUT) :: num_integers, num_large_counts, num_datatypes ombiner
<ol> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> </ol>	INTE	_GET_ENVELOPE(DATATYP COMBINER, IERROF GER DATATYPE, NUM_INT IERROR	E, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, A) EGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, YPE_GET_ENVELOPE returns information on the num-
46 47 48	ber and ty argument	ype of input arguments us s values returned can be	sed in the call that created the datatype. The number-of- used to provide sufficiently large arrays in the decoding ITS. This call and the meaning of the returned values is

described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

*Rationale.* By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

The decoded information keeps track of datatype duplications. This is important as one needs to distinguish between a predefined datatype and a dup of a predefined datatype. The former is a constant object that cannot be freed, while the latter is a derived datatype that can be freed. (*End of rationale.*)

The list in Table 5.1 has the values that can be returned in combiner on the left and the call associated with them on the right.

0		15
MPI_COMBINER_NAMED	a named predefined datatype	16
MPI_COMBINER_DUP	MPI_TYPE_DUP	17
MPI_COMBINER_CONTIGUOUS	MPI_TYPE_CONTIGUOUS	18
MPI_COMBINER_VECTOR	MPI_TYPE_VECTOR	19
MPI_COMBINER_HVECTOR	MPI_TYPE_CREATE_HVECTOR	20
MPI_COMBINER_INDEXED	MPI_TYPE_INDEXED	21
MPI_COMBINER_HINDEXED	MPI_TYPE_CREATE_HINDEXED	22
MPI_COMBINER_INDEXED_BLOCK	MPI_TYPE_CREATE_INDEXED_BLOCK	23
MPI_COMBINER_HINDEXED_BLOCK	MPI_TYPE_CREATE_HINDEXED_BLOCK	24
MPI_COMBINER_STRUCT	MPI_TYPE_CREATE_STRUCT	25
MPI_COMBINER_SUBARRAY	MPI_TYPE_CREATE_SUBARRAY	26
MPI_COMBINER_DARRAY	MPI_TYPE_CREATE_DARRAY	27
MPI_COMBINER_F90_REAL	MPI_TYPE_CREATE_F90_REAL	28
MPI_COMBINER_F90_COMPLEX	MPI_TYPE_CREATE_F90_COMPLEX	29
MPI_COMBINER_F90_INTEGER	MPI_TYPE_CREATE_F90_INTEGER	30
MPI_COMBINER_RESIZED	MPI_TYPE_CREATE_RESIZED	31
		32

Table 5.1: combiner values returned from MPI\_TYPE\_GET\_ENVELOPE

If combiner is MPI\_COMBINER\_NAMED then datatype is a named predefined datatype. If the MPI\_TYPE\_GET\_ENVELOPE variant without num\_large\_counts is invoked with a datatype that requires an output value of  $num\_large\_counts > 0$ , then an error of class MPI\_ERR\_TYPE is raised.

Rationale. The large count variant of this MPI procedure was added in MPI-4. It contains a new num\_large\_counts parameter. The other variant—the variant that existed before MPI-4—was not changed in order to preserve backwards compatibility. (End of rationale.)

The actual arguments used in the creation call for a datatype can be obtained using MPI\_TYPE\_GET\_CONTENTS.

MPI\_TYPE\_GET\_ENVELOPE and MPI\_TYPE\_GET\_CONTENTS also support large count types in separate additional MPI procedures in C (suffixed with the "\_c") and interface polymorphism in Fortran when using USE mpi\_f08.

1 2 3	MPI_TYPI	• • • •	max_integers, max_addresses, max_large_counts, _integers, array_of_addresses, rray_of_datatypes)	
4 5	IN	datatype	datatype to decode (handle)	
6 7	IN	max_integers	number of elements in array_of_integers (non-negative integer)	
8 9	IN	max_addresses	number of elements in array_of_addresses (non-negative integer)	
10 11 12 13	IN	max_large_counts	number of elements in array_of_large_counts (non-negative integer, only present for large count variants)	
14 15	IN	max_datatypes	number of elements in array_of_datatypes (non-negative integer)	
16 17	OUT	array_of_integers	contains integer arguments used in constructing datatype (array of integers)	
18 19 20	OUT	array_of_addresses	contains address arguments used in constructing datatype (array of integers)	
21 22 23	OUT	array_of_large_counts	contains large count arguments used in constructing datatype (array of integers, only present for large count variants)	
24 25 26	OUT	array_of_datatypes	contains datatype arguments used in constructing datatype (array of handles)	
27 28 29 30 31 32 33 34	C binding int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers, int max_addresses, int max_datatypes, int array_of_integers[], MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[]) int MPI_Type_get_contents_c(MPI_Datatype datatype, MPI_Count max_integers, MDI_Count max_integers,			
35 36 37 38	<pre>MPI_Count max_datatypes, int array_of_integers[], MPI_Aint array_of_addresses[], MPI_Count array_of_large_counts[], MPI_Datatype array_of_datatypes[])</pre>			
<ol> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> <li>47</li> <li>48</li> </ol>	MP1_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes,			

TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
	3
MPI_Type_get_contents(datatype, max_integers, max_addresses,	4
<pre>max_large_counts, max_datatypes, array_of_integers,</pre>	5
array_of_addresses, array_of_large_counts, array_of_datatypes,	6
ierror) !(_c)	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers,</pre>	9
<pre>max_addresses, max_large_counts, max_datatypes</pre>	10
<pre>INTEGER, INTENT(OUT) :: array_of_integers(max_integers)</pre>	11
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::	12
<pre>array_of_addresses(max_addresses)</pre>	13
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::	14
array_of_large_counts(max_large_counts)	15
TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
Fortran binding	18
MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	19
ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES,	20
IERROR)	21
INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,	22
ARRAY_OF_INTEGERS(*), ARRAY_OF_DATATYPES(*), IERROR	23
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)	24
INTEGER(KIND-FFI_ADDRESS_KIND) ARRAI_OF_ADDRESSES(*)	25
datatype must be a predefined unnamed or a derived datatype; the call is erroneous if	26
datatype is a predefined named datatype.	27
The values given for max_integers, max_addresses, max_large_counts, and	28
max_datatypes must be at least as large as the value returned in num_integers,	29
num_addresses, num_large_counts, and num_datatypes, respectively, in the call	30
MPI_TYPE_GET_ENVELOPE for the same datatype argument.	31
	32
<i>Rationale.</i> The arguments max_integers, max_addresses, max_large_counts, and	33
max_datatypes allow for error checking in the call. (End of rationale.)	34
	35
If the MPI_TYPE_GET_CONTENTS variant without max_large_counts is invoked with	36
a datatype that requires $> 0$ values in array_of_large_counts, then an error of class	37
MPI_ERR_TYPE is raised.	38
	39
Rationale. The large count variant of this MPI procedure was added in MPI-4.	40
It contains new max_large_counts and array_of_large_counts parameters. The other	41
variant—the variant that existed before MPI-4—was not changed in order to preserve	42
backwards compatibility. (End of rationale.)	43
The detetures not more of detetures and hardles to deteture 1. (1.)	43
The datatypes returned in array_of_datatypes are handles to datatype objects that	44
are equivalent to the datatypes used in the original construction call. If these were derived	45
datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPL TYPE FREE. If these were predefined datatypes, then	40
TO THE THE THE ALL THE ALL THE FACE IT THESE WERE DECEMBED CALLTANES THEN	֥

for freeing these datatypes with MPI\_TYPE\_FREE. If these were predefined datatypes, then 47the returned datatype is equal to that (constant) predefined datatype and cannot be freed. 48

1 2	The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is
3	undefined.
4	Note that MPI_TYPE_GET_CONTENTS can be invoked with a
5	datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL,
6	MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed
7	predefined datatype). In such a case, an empty array_of_datatypes is returned.
8	prodonnoù datatypo). In such a case, an ompty andy_or_datatypos is rotarnoù.
9	Rationale. The definition of datatype equivalence implies that equivalent predefined
10	datatypes are equal. By requiring the same handle for named predefined datatypes,
11	it is possible to use the == or .EQ. comparison operator to determine the datatype
12	involved. (End of rationale.)
13	
14	Advice to implementors. The datatypes returned in array_of_datatypes must appear
15	to the user as if each is an equivalent copy of the datatype used in the type constructor
16	call. Whether this is done by creating a new datatype or via another mechanism such
17	as a reference count mechanism is up to the implementation as long as the semantics
18	are preserved. (End of advice to implementors.)
19	
20	Rationale. The committed state and attributes of the returned datatype is delib-
21	erately left vague. The datatype used in the original construction may have been
22	modified since its use in the constructor call. Attributes can be added, removed, or
23	modified as well as having the datatype committed. The semantics given allow for
24	a reference count implementation without having to track these changes. (End of
25	rationale.)
26	In the deprecated datatype constructor calls, the address arguments in Fortran are
27	of type INTEGER. In the preferred calls, the address arguments are of type
28	INTEGER (KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all ad-
29	dresses in an argument of type INTEGER (KIND=MPI_ADDRESS_KIND). This is true even if the
30	deprecated calls were used. Thus, the location of values returned can be thought of as being
31	returned by the C bindings. It can also be determined by examining the preferred calls for
32	datatype constructors for the deprecated calls that involve addresses.
33	
34	Rationale. By having all address arguments returned in the
35	array_of_addresses argument, the result from a C and Fortran decoding of a datatype
36	gives the result in the same argument. It is assumed that an integer of type
37	INTEGER(KIND=MPI_ADDRESS_KIND) will be at least as large as the INTEGER argument
38	used in datatype construction with the old MPI-1 calls so no loss of information will
39	occur. (End of rationale.)
40	
41 42	The following defines what values are placed in each entry of the returned arrays
42	depending on the datatype constructor used for datatype. It also specifies the size of the
43 44	arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran,
45	the following calls were made:
46	PARAMETER (LARGE = 1000)
47	PARAMELER (LARGE = 1000)

```
<sup>47</sup> INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR
```

<sup>48</sup> INTEGER(KIND=MPI\_ADDRESS\_KIND) A(LARGE)

```
1
! CONSTRUCT DATATYPE TYPE (NOT SHOWN)
                                                                                        \mathbf{2}
CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)
                                                                                        3
IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
   WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, &
                                                                                        4
   " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
                                                                                        5
                                                                                        6
   CALL MPI_ABORT(MPI_COMM_WORLD, 99, IERROR)
ENDIF
CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)
                                                                                        9
or in C the analogous calls of:
                                                                                        10
                                                                                        11
#define LARGE 1000
                                                                                        12
int ni, na, nd, combiner, i[LARGE];
                                                                                        13
MPI_Aint a[LARGE];
                                                                                        14
MPI_Datatype type, d[LARGE];
                                                                                        15
/* construct datatype type (not shown) */
                                                                                        16
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
                                                                                        17
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
                                                                                        18
    fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
                                                                                        19
    fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d\n",
                                                                                        20
             LARGE);
                                                                                       21
    MPI_Abort(MPI_COMM_WORLD, 99);
                                                                                       22
};
                                                                                       23
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
                                                                                        24
The following describes the values of the arguments for each combiner. The lower case
                                                                                        25
name of arguments is used. Also, the descriptions below refer to MPI datatypes created
                                                                                        26
with procedures without large count arguments.
                                                                                        27
                                                                                        28
MPI_COMBINER_NAMED the datatype represent a predefined type and therefore it is er-
                                                                                       29
roneous to call MPI_TYPE_GET_CONTENTS.
                                                                                        30
MPI_COMBINER_DUP ni = 0, na = 0, nd = 1, and
                                                                                        31
                                                                                        32
                                                                                        33
                    Constructor argument
                                            С
                                                 Fortran location
                                                                                       34
                    oldtype
                                           d[0]
                                                      D(1)
                                                                                        35
MPI_COMBINER_CONTIGUOUS ni = 1, na = 0, nd = 1, and
                                                                                        36
                                                                                        37
                                                                                        38
                    Constructor argument
                                            С
                                                 Fortran location
                                                                                        39
                    count
                                           i[0]
                                                      I(1)
                                                                                        40
                                           d[0]
                                                      D(1)
                    oldtype
                                                                                        41
                                                                                        42
MPI_COMBINER_VECTOR ni = 3, na = 0, nd = 1, and
                                                                                        43
                                                                                        44
                    Constructor argument
                                            С
                                                 Fortran location
                                                                                        45
                                           i[0]
                    count
                                                      I(1)
                                                                                        46
                    blocklength
                                           i[1]
                                                      I(2)
                                                                                        47
                    stride
                                           i[2]
                                                      I(3)
```

d[0]

D(1)

oldtype

163

	Constructor argun	nent C Fort	ran location
	count	i[0]	I(1)
	blocklength	i[1]	I(2)
	stride	a[0]	A(1)
	oldtype	d[0]	D(1)
MPI COMBIN	ER_INDEXED ni = 2*co	unt+1. na = 0. 1	nd = 1, and
			·
	structor argument	C	Fortran locatio
coun		i[0]	I(1)
	_	[1] to i[i[0]]	I(2) to $I(I(1)+1)$
e			I(I(1)+2) to $I(2*I(1))$
oldty	vpe	d[0]	D(1)
MPI_COMBIN	ER_HINDEXED ni = com	unt+1, na = cou	nt, nd = 1, and
	Constructor argument	С	Fortran location
	count	i[0]	I(1)
	array_of_blocklengths	i[1] to i[i[0]]	I(2) to $I(I(1)+1)$
	array_of_displacements	a[0] to $a[i[0]-1]$	
	oldtype	d[0]	D(1)
	ER_INDEXED_BLOCK ni	= count+2, na	= 0, nd = 1, and
	Constructor argument	С	Fortran location
	Constructor argument count	C i[0]	Fortran location I(1)
	Constructor argument count blocklength	C i[0] i[1]	Fortran location I(1) I(2)
	Constructor argument count blocklength array_of_displacements	$\begin{array}{c} C \\ i[0] \\ i[1] \\ i[2] \text{ to } i[i[0]+1] \end{array}$	Fortran location I(1) I(2) I(3) to $I(I(1)+2)$
	Constructor argument count blocklength	C i[0] i[1]	Fortran location I(1) I(2)
	Constructor argument count blocklength array_of_displacements	$\begin{array}{c} C \\ i[0] \\ i[1] \\ i[2] \text{ to } i[i[0]+1] \\ d[0] \end{array}$	Fortran location I(1) I(2) I(3) to $I(I(1)+2)D(1)$
	Constructor argument count blocklength array_of_displacements oldtype	$\begin{array}{c} C \\ i[0] \\ i[1] \\ i[2] \text{ to } i[i[0]+1] \\ d[0] \end{array}$	Fortran location I(1) I(2) I(3) to $I(I(1)+2)D(1)$
	Constructor argument count blocklength array_of_displacements oldtype	$\frac{C}{i[0]} \\ i[1] \\ i[2] \text{ to } i[i[0]+1] \\ d[0] \\ \text{ni } = 2, \text{ na } = \text{ co}$	Fortran location I(1) I(2) I(3) to $I(I(1)+2)D(1)unt, nd = 1, and$
	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK	C i[0] i[1] i[2] to i[i[0]+1] d[0] ni = 2, na = co C	Fortran location I(1) I(2) I(3) to $I(I(1)+2)D(1)unt, nd = 1, andFortran location$
	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count	$     \frac{C}{i[0]} \\     i[1] \\     i[2] to i[i[0]+1] \\     d[0] \\     ni = 2, na = co \\     \hline     \hline     C \\     i[0]   $	Fortran location I(1) $I(2)$ $I(3)  to  I(I(1)+2)$ $D(1)$ unt, nd = 1, and Fortran location I(1)
	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count blocklength	$     \frac{C}{i[0]} \\     i[1] \\     i[2] to i[i[0]+1] \\     d[0] \\     ni = 2, na = co \\     \frac{C}{i[0]} \\     i[1]     $	Fortran location I(1) $I(2)$ $I(3)  to  I(I(1)+2)$ $D(1)$ unt, nd = 1, and Fortran location I(1) $I(2)$
MPI_COMBIN	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacements oldtype	$\frac{C}{i[0]}$ $i[1]$ $i[2] \text{ to } i[i[0]+1]$ $d[0]$ $ni = 2, na = co$ $\frac{C}{i[0]}$ $i[1]$ $a[0] \text{ to } a[i[0]-1]$ $d[0]$	Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) unt, nd = 1, and Fortran location I(1) I(2) A(1) to A(I(1)) D(1)
MPI_COMBIN	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacements	$\frac{C}{i[0]}$ $i[1]$ $i[2] \text{ to } i[i[0]+1]$ $d[0]$ $ni = 2, na = co$ $\frac{C}{i[0]}$ $i[1]$ $a[0] \text{ to } a[i[0]-1]$ $d[0]$	Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) unt, nd = 1, and Fortran location I(1) I(2) A(1) to A(I(1)) D(1)
MPI_COMBIN	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacements oldtype	$\frac{C}{i[0]}$ $i[1]$ $i[2] \text{ to } i[i[0]+1]$ $d[0]$ $ni = 2, na = co$ $\frac{C}{i[0]}$ $i[1]$ $a[0] \text{ to } a[i[0]-1]$ $d[0]$	Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) unt, nd = 1, and Fortran location I(1) I(2) A(1) to A(I(1)) D(1)
MPI_COMBIN	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacements oldtype ER_STRUCT ni = count	$\frac{C}{i[0]}$ $i[1]$ $i[2] \text{ to } i[i[0]+1]$ $d[0]$ ni = 2, na = co $\frac{C}{i[0]}$ $i[1]$ $a[0] \text{ to } a[i[0]-1]$ $d[0]$ $i=1, na = count, a =$	Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) unt, nd = 1, and Fortran location I(1) I(2) A(1) to A(I(1)) D(1) nd = count, and
MPI_COMBIN	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacements oldtype ER_STRUCT ni = count Constructor argument	$\frac{C}{i[0]} \\i[1]\\i[2] to i[i[0]+1]\\d[0]$ ni = 2, na = co $\frac{C}{i[0]} \\i[1]\\a[0] to a[i[0]-1]\\d[0]$ c+1, na = count,	Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) unt, nd = 1, and Fortran location I(1) I(2) A(1) to A(I(1)) D(1) nd = count, and Fortran location
MPI_COMBIN	Constructor argument count blocklength array_of_displacements oldtype ER_HINDEXED_BLOCK Constructor argument count blocklength array_of_displacements oldtype ER_STRUCT ni = count Constructor argument count	$\frac{C}{i[0]} \\i[1] \\i[2] to i[i[0]+1] \\d[0] \\d[0] \\d[0] \\d[0] \\c] \\c] \\c] \\c] \\c] \\c] \\c] \\c] \\c] \\c$	Fortran location I(1) I(2) I(3) to I(I(1)+2) D(1) unt, nd = 1, and Fortran location I(1) I(2) A(1) to A(I(1)) D(1) and = count, and Fortran location I(1)

Constructor argument	С	Fortran location
ndims	i[0]	I(1)
array_of_sizes	i[1] to $i[i[0]]$	I(2) to $I(I(1)+1)$
array_of_subsizes	i[i[0]+1] to $i[2*i[0]]$	I(I(1)+2) to $I(2*I(1)+1)$
array_of_starts	i[2*i[0]+1] to $i[3*i[0]]$	I(2*I(1)+2) to $I(3*I(1)+1)$
order	i[3*i[0]+1]	I(3*I(1)+2]
oldtype	d[0]	D(1)

#### MPI\_COMBINER\_SUBARRAY ni = 3\*ndims+2, na = 0, nd = 1, and

## MPI\_COMBINER\_DARRAY ni = 4\*ndims+4, na = 0, nd = 1, and

Constructor argument	С	Fortran location
size	i[0]	I(1)
rank	i[1]	I(2)
ndims	i[2]	I(3)
array_of_gsizes	i[3] to i[i[2]+2]	I(4) to $I(I(3)+3)$
array_of_distribs	i[i[2]+3] to $i[2*i[2]+2]$	I(I(3)+4) to $I(2*I(3)+3)$
array_of_dargs	i[2*i[2]+3] to $i[3*i[2]+2]$	I(2*I(3)+4) to $I(3*I(3)+3)$
$array_of_psizes$	i[3*i[2]+3] to $i[4*i[2]+2]$	I(3*I(3)+4) to $I(4*I(3)+3)$
order	i[4*i[2]+3]	I(4*I(3)+4)
oldtype	d[0]	D(1)

#### MPI\_COMBINER\_F90\_REAL ni = 2, na = 0, nd = 0, and

Constructor argument	С	Fortran location
р	i[0]	I(1)
r	i[1]	I(2)

#### MPI\_COMBINER\_F90\_COMPLEX ni = 2, na = 0, nd = 0, and

Constructor argument	С	Fortran location
р	i[0]	I(1)
r	i[1]	I(2)

## MPI\_COMBINER\_F90\_INTEGER ni = 1, na = 0, nd = 0, and

Constructor argument	С	Fortran location
r	i[0]	I(1)

#### MPI\_COMBINER\_RESIZED ni = 0, na = 2, nd = 1, and

Constructor argument	С	Fortran location
lb	a[0]	A(1)
extent	a[1]	A(2)
oldtype	d[0]	$\mathrm{D}(1)$

## 5.1.14 Examples

The following examples illustrate the use of derived datatypes.

 $46 \\ 47$ 

1 2	<b>Example 5.13</b> Send and receive a section of a 3D array.
3	REAL a(100,100,100), e(9,9,9)
4	INTEGER oneslice, twoslice, threeslice, myrank, ierr
5	INTEGER(KIND=MPI_ADDRESS_KIND) lb, sizeofreal
6	INTEGER status(MPI_STATUS_SIZE)
7	
8	! extract the section a(1:17:2, 3:11, 2:10)
9 10	! and store it in e(:,:,:).
10	
12	CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
13	CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
14	
15	! create datatype for a 1D section
16	CALL MPI_TYPE_VECTOR(9, 1, 2, MPI_REAL, oneslice, ierr)
17	
18 19	! create datatype for a 2D section
20	CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*sizeofreal, oneslice, &
21	twoslice, ierr)
22	! create datatype for the entire section
23	CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice, &
24	threeslice, ierr)
25	
26	CALL MPI_TYPE_COMMIT(threeslice, ierr)
27 28	CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, &
29	MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
30	
31	<b>Example 5.14</b> Copy the (strictly) lower triangular part of a matrix.
32	REAL a(100,100), b(100,100)
33	INTEGER disp(100), blocklen(100), ltype, myrank, ierr
34	INTEGER status(MPI_STATUS_SIZE)
35 36	
37	! copy lower triangular part of array a
38	! onto lower triangular part of array b
39	CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
40	CALL MIT_COMM_MARK(MIT_COMM_WORLD, MyTalk, Tell)
41	! compute start and size of each column
42	DO i=1,100
43 44	disp(i) = 100*(i-1) + i
44 45	blocklen(i) = 100-i
46	END DO
47	L speate deteture for lower trier miles sent
48	! create datatype for lower triangular part

CALL MPI\_TYPE\_INDEXED(100, blocklen, disp, MPI\_REAL, ltype, ierr) CALL MPI\_TYPE\_COMMIT(ltype, ierr) CALL MPI\_SENDRECV(a, 1, ltype, myrank, 0, b, 1, & ltype, myrank, 0, MPI\_COMM\_WORLD, status, ierr)

**Example 5.15** Transpose a matrix. REAL a(100,100), b(100,100) INTEGER row, xpose, myrank, ierr INTEGER(KIND=MPI\_ADDRESS\_KIND) lb, sizeofreal INTEGER status(MPI\_STATUS\_SIZE) ! transpose matrix a onto b CALL MPI\_COMM\_RANK(MPI\_COMM\_WORLD, myrank, ierr) CALL MPI\_TYPE\_GET\_EXTENT(MPI\_REAL, lb, sizeofreal, ierr) ! create datatype for one row CALL MPI\_TYPE\_VECTOR(100, 1, 100, MPI\_REAL, row, ierr) ! create datatype for matrix in row-major order CALL MPI\_TYPE\_CREATE\_HVECTOR(100, 1, sizeofreal, row, xpose, ierr) CALL MPI\_TYPE\_COMMIT(xpose, ierr) ! send matrix in row-major order and receive in column major order CALL MPI\_SENDRECV(a, 1, xpose, myrank, 0, b, 100\*100, & MPI\_REAL, myrank, 0, MPI\_COMM\_WORLD, status, ierr)

**Example 5.16** Another approach to the transpose problem:

REAL a(100,100), b(100,100) INTEGER row, row1 INTEGER(KIND=MPI\_ADDRESS\_KIND) lb, sizeofreal INTEGER myrank, ierr INTEGER status(MPI\_STATUS\_SIZE)

CALL MPI\_COMM\_RANK(MPI\_COMM\_WORLD, myrank, ierr)

! transpose matrix a onto b

CALL MPI\_TYPE\_GET\_EXTENT(MPI\_REAL, lb, sizeofreal, ierr)

! create datatype for one row CALL MPI\_TYPE\_VECTOR(100, 1, 100, MPI\_REAL, row, ierr) 167

2 3

4

5

6 7

9

10

11

12

13 14

15 16

17 18

19 20

21

22 23

24

25 26

27 28

29

30

31 32 33

34

35

36

37

38

39 40

41 42

43 44

45 46

47

```
1
     ! create datatype for one row, with the extent of one real number
2
     1b = 0
3
     CALL MPI_TYPE_CREATE_RESIZED(row, lb, sizeofreal, row1, ierr)
4
5
     CALL MPI_TYPE_COMMIT(row1, ierr)
6
7
     ! send 100 rows and receive in column major order
8
     CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100, &
9
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
10
11
12
     Example 5.17 Use of MPI datatypes to manipulate an array of structures.
13
14
     struct Partstruct
15
     ſ
16
                type; /* particle type */
        int
17
        double d[6]; /* particle coordinates */
18
                       /* some additional information */
        char
               b[7];
19
     };
20
21
     struct Partstruct
                           particle[1000];
22
23
     int
                   i, dest, tag;
^{24}
     MPI_Comm
                   comm;
25
26
27
     /* build datatype describing structure */
28
29
     MPI_Datatype Particlestruct, Particletype;
30
     MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
^{31}
                   blocklen[3] = \{1, 6, 7\};
     int
32
     MPI_Aint
                   disp[3];
33
     MPI_Aint
                   base, lb, sizeofentry;
34
35
36
     /* compute displacements of structure components */
37
38
     MPI_Get_address(particle, disp);
39
     MPI_Get_address(particle[0].d, disp+1);
40
     MPI_Get_address(particle[0].b, disp+2);
41
     base = disp[0];
42
     for (i=0; i < 3; i++) disp[i] = MPI_Aint_diff(disp[i], base);</pre>
43
44
     MPI_Type_create_struct(3, blocklen, disp, type, &Particlestruct);
45
46
     /* Since the compiler may pad the structure, it is best to explicitly
47
        set the extent of the MPI datatype for a structure element using
48
```

```
MPI_Type_create_resized */
/* compute extent of the structure */
MPI_Get_address(particle+1, &sizeofentry);
sizeofentry = MPI_Aint_diff(sizeofentry, base);
/* build datatype describing structure */
MPI_Type_create_resized(Particlestruct, 0, sizeofentry, &Particletype);
/* 4.1: send the entire array */
MPI_Type_commit(&Particletype);
MPI_Send(particle, 1000, Particletype, dest, tag, comm);
/* 4.2: send only the entries of type zero particles,
        preceded by the number of such entries */
MPI_Datatype Zparticles;
                           /* datatype describing all particles
                              with type zero (needs to be recomputed
                               if types change) */
MPI_Datatype Ztype;
             zdisp[1000];
int
int
             zblock[1000], j, k;
int
             zzblock[2] = \{1,1\};
MPI_Aint
             zzdisp[2];
MPI_Datatype zztype[2];
/* compute displacements of type zero particles */
i = 0;
for (i=0; i < 1000; i++)
   if (particle[i].type == 0)
      {
        zdisp[j] = i;
        zblock[j] = 1;
        j++;
      }
/* create datatype for type zero particles */
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
/* prepend particle count */
MPI_Get_address(&j, zzdisp);
MPI_Get_address(particle, zzdisp+1);
zztype[0] = MPI_INT;
```

1

2 3

4

5

6 7

8

9 10 11

12 13

14

15 16 17

18

19 20

21

22

23

24 25

26

27

28

29

30 31

32

33

34

35

36

37

38

39

40 41

42

43 44

45

46

47

```
1
     zztype[1] = Zparticles;
\mathbf{2}
     MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
3
4
     MPI_Type_commit(&Ztype);
\mathbf{5}
     MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
6
\overline{7}
8
     /* A probably more efficient way of defining Zparticles */
9
10
     /* consecutive particles with index zero are handled as one block */
11
     i=0;
12
     for (i=0; i < 1000; i++)</pre>
13
        if (particle[i].type == 0)
14
            ſ
15
               for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
16
               zdisp[j] = i;
17
               zblock[j] = k-i;
18
               j++;
19
               i = k;
20
            }
21
     MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
22
23
^{24}
     /* 4.3: send the first two coordinates of all entries */
25
26
     MPI_Datatype Allpairs;
                                   /* datatype for all pairs of coordinates */
27
28
     MPI_Type_get_extent(Particletype, &lb, &sizeofentry);
29
30
     /* sizeofentry can also be computed by subtracting the address
^{31}
        of particle[0] from the address of particle[1] */
32
33
     MPI_Type_create_hvector(1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
34
     MPI_Type_commit(&Allpairs);
35
     MPI_Send(particle[0].d, 1, Allpairs, dest, tag, comm);
36
37
     /* an alternative solution to 4.3 */
38
39
     MPI_Datatype Twodouble;
40
41
     MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);
42
43
     MPI_Datatype Onepair;
                               /* datatype for one pair of coordinates, with
44
                                 the extent of one particle entry */
45
46
     MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
47
     MPI_Type_commit(&Onepair);
48
```

MPI\_Send(particle[0].d, 1000, Onepair, dest, tag, comm);

```
Example 5.18 The same manipulations as in the previous example, but use absolute
addresses in datatypes.
struct Partstruct
ſ
    int
           type;
    double d[6];
    char
           b[7];
};
struct Partstruct particle[1000];
/* build datatype describing first array entry */
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
             block[3] = \{1, 6, 7\};
int
MPI_Aint
             disp[3];
MPI_Get_address(particle, disp);
MPI_Get_address(particle[0].d, disp+1);
MPI_Get_address(particle[0].b, disp+2);
MPI_Type_create_struct(3, block, disp, type, &Particletype);
/* Particletype describes first array entry -- using absolute
   addresses */
/* 5.1: send the entire array */
MPI_Type_commit(&Particletype);
MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);
/* 5.2: send the entries of type zero,
        preceded by the number of such entries */
MPI_Datatype Zparticles, Ztype;
int
             zdisp[1000];
int
             zblock[1000], i, j, k;
             zzblock[2] = \{1,1\};
int
MPI_Datatype zztype[2];
MPI_Aint
             zzdisp[2];
j=0;
```

1

2 3

4

5 6

7

8

9

10

11

12 13

14 15

16 17

18

19

20

21 22

23

24

25

26 27

28

29 30

31 32

33

34 35 36

37

38 39

40 41

42

43

44

45

 $46 \\ 47$ 

```
1
     for (i=0; i < 1000; i++)</pre>
\mathbf{2}
          if (particle[i].type == 0)
3
              {
4
                  for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);</pre>
5
                  zdisp[j] = i;
6
                  zblock[j] = k-i;
7
                   j++;
8
                   i = k;
9
              }
10
     MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
11
     /* Zparticles describe particles with type zero, using
12
         their absolute addresses*/
13
14
     /* prepend particle count */
15
     MPI_Get_address(&j, zzdisp);
16
     zzdisp[1] = (MPI_Aint)0;
17
     zztype[0] = MPI_INT;
18
     zztype[1] = Zparticles;
19
     MPI_Type_create_struct(2, zzblock, zzdisp, zztype, &Ztype);
20
21
     MPI_Type_commit(&Ztype);
22
     MPI_Send(MPI_BOTTOM, 1, Ztype, dest, tag, comm);
23
^{24}
     Example 5.19 This example shows how datatypes can be used to handle unions.
25
26
     union {
27
         int
                 ival;
28
                 fval;
         float
29
            } u[1000];
30
^{31}
              i, utype;
     int
32
33
     /* All entries of u have identical type; variable
34
         utype keeps track of their current type */
35
36
     MPI_Datatype
                      mpi_utype[2];
37
     MPI_Aint
                      ubase, extent;
38
39
     /* compute an MPI datatype for each possible union type;
40
         assume values are left-aligned in union storage. */
41
42
     MPI_Get_address(u, &ubase);
43
     MPI_Get_address(u+1, &extent);
44
     extent = MPI_Aint_diff(extent, ubase);
45
46
     MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);
47
48
```

```
MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);
for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);
/* actual communication */
MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

**Example 5.20** This example shows how a datatype can be decoded. The routine printdatatype prints out the elements of the datatype. Note the use of MPI\_Type\_free for datatypes that are not predefined.

```
/*
  Example of decoding a datatype.
  Returns 0 if the datatype is predefined, 1 otherwise
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int printdatatype(MPI_Datatype datatype)
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;
    MPI_Type_get_envelope(datatype,
                          &num_ints, &num_adds, &num_dtypes, &combiner);
    switch (combiner) {
    case MPI_COMBINER_NAMED:
        printf("Datatype is named:");
        /* To print the specific type, we can match against the
           predefined forms. We can NOT use a switch statement here
           We could also use MPI_TYPE_GET_NAME if we prefered to use
           names that the user may have changed.
         */
        if
                (datatype == MPI_INT)
                                          printf("MPI_INT\n");
        else if (datatype == MPI_DOUBLE) printf("MPI_DOUBLE\n");
        ... else test for other types ...
        return 0;
        break;
    case MPI_COMBINER_STRUCT:
    case MPI_COMBINER_STRUCT_INTEGER:
        printf("Datatype is struct containing");
                        = (int *)malloc(num_ints * sizeof(int));
        array_of_ints
        array_of_adds
                        =
```

1

2 3

4 5

6

7

9

10

11 12

13

14 15

16

17

18

19

20

21

22

23

24

25

26

27 28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

1	(MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));
2	array_of_dtypes = (MPI_Datatype *)
3	<pre>malloc(num_dtypes * sizeof(MPI_Datatype));</pre>
4	MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
5	array_of_ints, array_of_adds, array_of_dtypes);
6	printf(" %d datatypes:\n", array_of_ints[0]);
7	for (i=0; i <array_of_ints[0]; i++)="" th="" {<=""></array_of_ints[0];>
8	printf("blocklength %d, displacement %ld, type:\n",
9	array_of_ints[i+1], (long)array_of_adds[i]);
10	if (printdatatype(array_of_dtypes[i])) {
11	/* Note that we free the type ONLY if it
12	is not predefined */
13	<pre>MPI_Type_free(&amp;array_of_dtypes[i]);</pre>
14	}
15	}
16	free(array_of_ints);
17	free(array_of_adds);
18	free(array_of_dtypes);
19	break;
20	other combiner values
21	default:
22	printf("Unrecognized combiner type\n");
23	}
24	return 1;
25	}
26	

## 5.2 Pack and Unpack

29Some existing communication libraries provide pack/unpack functions for sending noncon-30 tiguous data. In these, the user explicitly packs data into a contiguous buffer before sending  $^{31}$ it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are 32 described in Section 5.1, allow one, in most cases, to avoid explicit packing and unpacking. 33 The user specifies the layout of the data to be sent or received, and the communication 34library directly accesses a noncontiguous buffer. The pack/unpack routines are provided 35 for compatibility with previous libraries. Also, they provide some functionality that is not 36 otherwise available in MPI. For instance, a message can be received in several parts, where 37 the receive operation done on a later part may depend on the content of a former part. 38 Another use is that outgoing messages may be explicitly buffered in user supplied space, 39 thus overriding the system buffering policy. Finally, the availability of pack and unpack 40 operations facilitates the development of additional communication libraries layered on top 41 of MPI. 42

43

27

- 44
- 45
- 46
- 47
- 48

MPI_PACH	K(inbuf, incount, datatype, out	buf, outsize, position, comm)	1
IN	inbuf	input buffer start (choice)	2
IN	incount	number of input data items (non-negative integer)	3 4
IN	datatype	datatype of each input data item (handle)	5
OUT	outbuf	output buffer start (choice)	6
IN	outsize	output buffer size, in bytes (non-negative integer)	7
			8 9
INOUT	position	current position in buffer, in bytes (integer)	10
IN	comm	communicator for packed message (handle)	11
C binding	ar.		12
		nt incount, MPI_Datatype datatype,	13
		stsize, int *position, MPI_Comm comm)	14 15
int MPT F	Pack c(const void *inbuf	MPI_Count incount, MPI_Datatype datatype,	16
1110 111 1_1		ount outsize, MPI_Count *position,	17
	MPI_Comm comm)	-	18
Fortran 2	2008 binding		19 20
	0	, outbuf, outsize, position, comm, ierror)	20 21
	(*), DIMENSION(), INTEN		22
	ER, INTENT(IN) :: incoun		23
	<pre>[MPI_Datatype), INTENT(IN (*), DIMENSION() :: out</pre>		24
	ER, INTENT(INOUT) :: pos		25 26
	(MPI_Comm), INTENT(IN) ::		20
INTEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	28
MPI_Pack(	inbuf, incount, datatype	, outbuf, outsize, position, comm, ierror)	29
	!(_c)	-	30
	(*), DIMENSION(), INTEN		31 32
		INTENT(IN) :: incount, outsize	33
	<pre>[MPI_Datatype), INTENT(IN (*), DIMENSION() :: out</pre>	· -	34
		INTENT(INOUT) :: position	35
TYPE(	(MPI_Comm), INTENT(IN) ::	comm	36
INTEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	37 38
Fortran k	binding		39
		, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)	40
• -	> INBUF(*), OUTBUF(*)		41
TNLEC	ER INCOUNT, DATATYPE, UU	TSIZE, POSITION, COMM, IERROR	42
	0	r specified by inbuf, incount, datatype into the buffer	43 44
	0	The input buffer can be any communication buffer	45
anowed in	INFI_SEND. The output but	fer is a contiguous storage area containing outsize	

space specified by outbuf and outsize. The input buffer can be any communication buffer allowed in MPI\_SEND. The output buffer is a contiguous storage area containing outsize bytes, starting at the address outbuf (length is counted in *bytes*, not elements, as if it were a communication buffer for a message of type MPI\_PACKED). The input value of **position** is the first location in the output buffer to be used for packing. **position** is incremented by the size of the packed message, and the output value of **position** is the first location in the output buffer following the locations occupied by the packed message. The **comm** argument is the communicator that will be subsequently used for sending the packed message.

```
MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)
8
9
       IN
                 inbuf
                                            input buffer start (choice)
10
       IN
                insize
                                            size of input buffer, in bytes (non-negative integer)
11
                position
       INOUT
                                            current position in bytes (integer)
12
       OUT
                outbuf
                                            output buffer start (choice)
13
14
       IN
                outcount
                                            number of items to be unpacked (integer)
15
       IN
                datatype
                                            datatype of each output data item (handle)
16
       IN
                comm
                                            communicator for packed message (handle)
17
18
19
     C binding
     int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,
20
21
                    int outcount, MPI_Datatype datatype, MPI_Comm comm)
22
     int MPI_Unpack_c(const void *inbuf, MPI_Count insize, MPI_Count *position,
23
                    void *outbuf, MPI_Count outcount, MPI_Datatype datatype,
24
                    MPI_Comm comm)
25
26
     Fortran 2008 binding
27
     MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
                    ierror)
28
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
29
30
          INTEGER, INTENT(IN) :: insize, outcount
         INTEGER, INTENT(INOUT) :: position
31
32
         TYPE(*), DIMENSION(...) :: outbuf
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
37
                    ierror) !(_c)
38
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
39
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
40
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
41
         TYPE(*), DIMENSION(..) :: outbuf
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
48
                    IERROR)
```

1

 $\mathbf{2}$ 

3

4

5

<type></type>	INBUF(*)	, OUTBUF(*	)			
INTEGER	INSIZE,	POSITION,	OUTCOUNT	, DATATYPE,	COMM,	IERROR

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI\_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

Advice to users. Note the difference between MPI\_RECV and MPI\_UNPACK: in MPI\_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI\_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined a priori. (End of advice to users.)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or sscanf in C, for a similar function.)

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI\_PACK, where the first call provides position = 0, and each successive call inputs the value of **position** that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI\_PACKED. Any point-to-point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI\_PACKED.

A message sent with any type (including MPI\_PACKED) can be received using the type MPI\_PACKED. Such a message can then be unpacked by calls to MPI\_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI\_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a <sup>47</sup> substring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two <sup>48</sup>

1 2 3

4

5

6

 $\overline{7}$ 

8

9

10

11 12

13

14

15

16

17

18

19

20

21

22 23

 $^{24}$ 

25

26

27 28

29

30

 $^{31}$ 

32 33

34

35 36

37

38

39

40 41

42

43 44

 $45 \\ 46$ 

packing units and then unpack the result as one packing unit; nor can one unpack a substring
 of a packing unit as a separate packing unit. Each packing unit, that was created by a related
 sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of
 related unpack calls.
 *Rationale.* The restriction on "atomic" packing and unpacking of packing units
 allows the implementation to add at the head of packing units additional information,

such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (*End of rationale.*)

- The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.
- <sup>14</sup> MPI\_PACK\_SIZE(incount, datatype, comm, size)

```
IN
                 incount
                                            count argument to packing call (non-negative integer)
16
17
       IN
                datatype
                                            datatype argument to packing call (handle)
18
       IN
                comm
                                            communicator argument to packing call (handle)
19
       OUT
                size
                                            upper bound on size of packed message, in bytes
20
                                            (non-negative integer)
21
22
23
     C binding
^{24}
     int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
25
                    int *size)
26
     int MPI_Pack_size_c(MPI_Count incount, MPI_Datatype datatype,
27
                    MPI_Comm comm, MPI_Count *size)
28
29
     Fortran 2008 binding
30
     MPI_Pack_size(incount, datatype, comm, size, ierror)
^{31}
          INTEGER, INTENT(IN) :: incount
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         INTEGER, INTENT(OUT) :: size
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Pack_size(incount, datatype, comm, size, ierror) !(_c)
37
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
41
```

- <sup>41</sup> INTEGER, OPTIONAL, INTENT(OUT) :: ierror
- 43 Fortran binding

```
    <sup>44</sup> MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    <sup>45</sup> INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

<sup>47</sup> A call to MPI\_PACK\_SIZE(incount, datatype, comm, size) returns in size an upper bound <sup>48</sup> on the increment in position that is effected by a call to MPI\_PACK(inbuf, incount, datatype,

8

9 10

11

12 13

outbuf, outcount, position, comm). If the packed size of the datatype cannot be expressed by the size parameter, then MPI\_PACK\_SIZE sets the value of size to MPI\_UNDEFINED.

*Rationale.* The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

```
Example 5.21 An example using MPI_PACK.
int
           position, i, j, a[2];
           buff[1000];
char
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
    /* SENDER CODE */
    position = 0;
    MPI_Pack(&i, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
    MPI_Pack(&j, 1, MPI_INT, buff, 1000, &position, MPI_COMM_WORLD);
    MPI_Send(buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else /* RECEIVER CODE */
    MPI_Recv(a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
Example 5.22 An elaborate example.
      position, i = 200;
int
float a[200];
char buff[1000]; /* larger than or equal to the size returned
                     from MPI_PACK_SIZE for 1,newtype */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
    /* SENDER CODE */
    int len[2];
    MPI_Aint disp[2];
    MPI_Datatype type[2], newtype;
    /* build datatype for i followed by a[0]...a[i-1] */
    len[0] = 1;
    len[1] = i;
    MPI_Get_address(&i, disp);
    MPI_Get_address(a, disp+1);
    type[0] = MPI_INT;
    type[1] = MPI_FLOAT;
```

1

2

3

4

5

6 7

8 9

10

11 12

13

14

15

16 17

18

19

20

21

22

23

24 25

26 27

28

29

30

31

32

33

34

35 36

37

38

39 40

41 42

43

44

45

46

47

```
MPI_Type_create_struct(2, len, disp, type, &newtype);
    MPI_Type_commit(&newtype);
    /* Pack i followed by a[0]...a[i-1]*/
   position = 0;
    MPI_Pack(MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);
    /* Send */
    MPI_Send(buff, position, MPI_PACKED, 1, 0,
             MPI_COMM_WORLD);
/* ****
   One can replace the last three lines with
   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   **** */
}
else if (myrank == 1)
{
    /* RECEIVER CODE */
    MPI_Status status;
    /* Receive */
   MPI_Recv(buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
    /* Unpack i */
    position = 0;
    MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);
    /* Unpack a[0]...a[i-1] */
    MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

**Example 5.23** Each process sends a count, followed by count characters to the root; the root concatenates all characters into one string.

 $\mathbf{2}$ 

 $\frac{24}{25}$ 

```
/* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);
      /* pack count, followed by count characters */
position = 0;
MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
if (myrank != root) {
    /* gather at root sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, NULL, 0,
               MPI_DATATYPE_NULL, root, comm);
    /* gather at root packed messages */
    MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
                NULL, NULL, MPI_DATATYPE_NULL, root, comm);
         /* root code */
} else {
    /* gather sizes of all packed messages */
    MPI_Gather(&position, 1, MPI_INT, counts, 1,
               MPI_INT, root, comm);
    /* gather all packed messages */
    displs[0] = 0;
    for (i=1; i < gsize; i++)</pre>
        displs[i] = displs[i-1] + counts[i-1];
    totalcount = displs[gsize-1] + counts[gsize-1];
    rbuf = (char *)malloc(totalcount);
    cbuf = (char *)malloc(totalcount);
    MPI_Gatherv(lbuf, position, MPI_PACKED, rbuf,
                counts, displs, MPI_PACKED, root, comm);
    /* unpack all messages and concatenate strings */
    concat_pos = 0;
    for (i=0; i < gsize; i++) {</pre>
        position = 0;
        MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
                   &position, &count, 1, MPI_INT, comm);
        MPI_Unpack(rbuf+displs[i], totalcount-displs[i],
                   &position, cbuf+concat_pos, count, MPI_CHAR, comm);
        concat_pos += count;
    }
    cbuf[concat_pos] = '\0';
}
```

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9

10

11 12

13

14

15

16 17

18

19

20 21

22

23

24

25 26

27

28

29

30

31

32

33

34

35 36

37

38

39

40

41

42

43

44

45

46

47

```
5.3
            Canonical MPI_PACK and MPI_UNPACK
1
\mathbf{2}
     These functions read/write data to/from the buffer in the "external32" data format specified
3
     in Section 14.5.2, and calculate the size needed for packing. Their first arguments specify
4
     the data format, for future extensibility, but currently the only valid value of the datarep
5
     argument is "external32".
6
7
           Advice to users. These functions could be used, for example, to send typed data in a
8
           portable format from one MPI implementation to another. (End of advice to users.)
9
10
         The buffer will contain exactly the packed data, without headers. MPI_BYTE should
11
     be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
12
           Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message
13
           and further specifies the exact format of the data. Since MPI_PACK may (and is
14
           allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed
15
           with MPI_PACK_EXTERNAL. (End of rationale.)
16
17
18
19
     MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)
20
       IN
                 datarep
                                             data representation (string)
21
22
                                             input buffer start (choice)
       IN
                 inbuf
23
       IN
                 incount
                                             number of input data items (integer)
24
       IN
                 datatype
                                             datatype of each input data item (handle)
25
26
       OUT
                 outbuf
                                             output buffer start (choice)
27
       IN
                 outsize
                                             output buffer size, in bytes (integer)
28
       INOUT
                 position
                                             current position in buffer, in bytes (integer)
29
30
^{31}
     C binding
32
     int MPI_Pack_external(const char datarep[], const void *inbuf, int incount,
33
                     MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
34
                     MPI_Aint *position)
35
     int MPI_Pack_external_c(const char datarep[], const void *inbuf,
36
                     MPI_Count incount, MPI_Datatype datatype, void *outbuf,
37
                     MPI_Count outsize, MPI_Count *position)
38
39
     Fortran 2008 binding
40
     MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
41
                     position, ierror)
42
          CHARACTER(LEN=*), INTENT(IN) :: datarep
43
          TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
44
          INTEGER, INTENT(IN) :: incount
45
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
          TYPE(*), DIMENSION(...) :: outbuf
47
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
48
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
```

INTEC	ER, OPTIONAL, INTEN	T(OUT) :: ierror	1	
MPI_Pack_	external(datarep, i	nbuf, incount, datatype, outbuf, outsize,	2 3	
position, ierror) !(_c)				
	CHARACTER(LEN=*), INTENT(IN) :: datarep			
	(*), DIMENSION(),		6	
		IND), INTENT(IN) :: incount, outsize	7	
	(MPI_Datatype), INTE	01	8	
	(*), DIMENSION() :	: outbur IND), INTENT(INOUT) :: position	9	
	ER, OPTIONAL, INTEN	• •	10	
			11	
Fortran h			12 13	
MPI_PACK_		NBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,	13 14	
CILAD	POSITION, IERRO ACTER*(*) DATAREP	IR)	14	
	> INBUF(*), OUTBUF(	*)	16	
• 1	ER INCOUNT, DATATYP		17	
	•	_KIND) OUTSIZE, POSITION	18	
		_ , _ , _ , _ , _ , _ ,	19	
			20	
MPI_UNP/	ACK_EXTERNAL(datare	p, inbuf, insize, position, outbuf, outcount, datatype)	21	
IN	datarep	data representation (string)	22	
	·	- (	23 24	
IN	inbuf	input buffer start (choice)	24 25	
IN	insize	input buffer size, in bytes (integer)	26	
INOUT	position	current position in buffer, in bytes (integer)	27	
OUT	outbuf	output buffer start (choice)	28	
IN	outcount	number of output data items (integer)	29 30	
IN	datatype	datatype of output data item (handle)	30 31	
	51		32	
C bindin	g		33	
	0	t char datarep[], const void *inbuf,	34	
	-	e, MPI_Aint *position, void *outbuf,	35	
	int outcount, M	PI_Datatype datatype)	36	
int MPT I	Innack external c(co	<pre>nst char datarep[], const void *inbuf,</pre>	37	
1110 111 1_0	-	e, MPI_Count *position, void *outbuf,	38	
		punt, MPI_Datatype datatype)	39	
Fortage (			40	
	2008 binding	inhuf incide position outbuf outcount	41 42	
<pre>MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,</pre>				
CHARACTER(LEN=*), INTENT(IN) :: datarep				
	(*), DIMENSION(),	-	45	
		_KIND), INTENT(IN) :: insize	46	
INTEC	ER(KIND=MPI_ADDRESS	_KIND), INTENT(INOUT) :: position	47	
TYPE(*), DIMENSION() :: outbuf				

```
1
         INTEGER, INTENT(IN) :: outcount
\mathbf{2}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
5
                   datatype, ierror) !(_c)
6
         CHARACTER(LEN=*), INTENT(IN) :: datarep
7
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
9
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
10
         TYPE(*), DIMENSION(..) :: outbuf
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
16
                   DATATYPE, IERROR)
17
         CHARACTER*(*) DATAREP
18
         <type> INBUF(*), OUTBUF(*)
19
         INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
20
         INTEGER OUTCOUNT, DATATYPE, IERROR
21
22
23
     MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)
^{24}
       IN
                datarep
                                           data representation (string)
25
26
       IN
                incount
                                           number of input data items (integer)
27
       IN
                datatype
                                           datatype of each input data item (handle)
28
       OUT
                size
                                           output buffer size, in bytes (integer)
29
30
     C binding
^{31}
     int MPI_Pack_external_size(const char datarep[], int incount,
32
                   MPI_Datatype datatype, MPI_Aint *size)
33
34
     int MPI_Pack_external_size_c(const char datarep[], MPI_Count incount,
35
                   MPI_Datatype datatype, MPI_Count *size)
36
37
     Fortran 2008 binding
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
38
         CHARACTER(LEN=*), INTENT(IN) :: datarep
39
         INTEGER, INTENT(IN) :: incount
40
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror) !(_c)
45
         CHARACTER(LEN=*), INTENT(IN) :: datarep
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
Fouture binding	3
Fortran binding MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)	4 5
CHARACTER*(*) DATAREP	6
INTEGER INCOUNT, DATATYPE, IERROR	7
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE	8
	9
	10
	11
	12
	13
	14
	15
	16
	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31
	32
	33
	34
	35 36
	30
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

## Chapter 6

# **Collective Communication**

## 6.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

• MPI\_BARRIER, MPI\_IBARRIER, MPI\_BARRIER\_INIT: Barrier synchronization across all members of a group (Section 6.3, Section 6.12.1, and Section 6.13.1).

 $^{24}$ 

 $45 \\ 46$ 

- MPI\_BCAST, MPI\_IBCAST, MPI\_BCAST\_INIT: Broadcast from one member to all members of a group (Section 6.4, Section 6.12.2, and Section 6.13.2). This is shown as "broadcast" in Figure 6.1.
- MPI\_GATHER, MPI\_IGATHER, MPI\_GATHER\_INIT, MPI\_GATHERV, MPI\_IGATHERV, MPI\_GATHERV\_INIT, : Gather data from all members of a group to one member (Section 6.5, Section 6.12.3, and Section 6.13.3). This is shown as "gather" in Figure 6.1.
- MPI\_SCATTER, MPI\_ISCATTER, MPI\_SCATTER\_INIT, MPI\_SCATTERV, MPI\_ISCATTERV, MPI\_SCATTERV\_INIT: Scatter data from one member to all members of a group (Section 6.6, Section 6.12.4, and Section 6.13.4). This is shown as "scatter" in Figure 6.1.
- MPI\_ALLGATHER, MPI\_IALLGATHER, MPI\_ALLGATHER\_INIT, MPI\_ALLGATHERV, MPI\_IALLGATHERV, MPI\_ALLGATHERV\_INIT: A variation on Gather where all members of a group receive the result (Section 6.7, Section 6.12.5, and Section 6.13.5). This is shown as "allgather" in Figure 6.1.
- MPI\_ALLTOALL, MPI\_IALLTOALL, MPI\_ALLTOALL\_INIT, MPI\_ALLTOALLV, MPI\_IALLTOALLV, MPI\_ALLTOALLV\_INIT, MPI\_ALLTOALLW, MPI\_IALLTOALLW, MPI\_ALLTOALLW\_INIT: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 6.8, Section 6.12.6, and Section 6.13.6). This is shown as "complete exchange" in Figure 6.1.
- MPI\_ALLREDUCE, MPI\_IALLREDUCE, MPI\_ALLREDUCE\_INIT, MPI\_REDUCE, MPI\_IREDUCE, MPI\_REDUCE\_INIT: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 6.9.6, Section 6.12.8, and Section 6.13.8) and a variation where the result is returned to only one member (Section 6.9, Section 6.12.7, and Section 6.13.7).

• MPI\_REDUCE\_SCATTER\_BLOCK, MPI\_IREDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER\_BLOCK\_INIT, MPI\_REDUCE\_SCATTER, MPI\_IREDUCE\_SCATTER, MPI\_REDUCE\_SCATTER\_INIT: A combined reduction and scatter operation (Section 6.10, Section 6.12.9, Section 6.12.10, Section 6.13.9, and Section 6.13.10).

• MPI\_SCAN, MPI\_ISCAN, MPI\_SCAN\_INIT, MPI\_EXSCAN, MPI\_IEXSCAN,

MPI\_EXSCAN\_INIT: Scan across all members of a group (also called prefix) (Section 6.11, Section 6.11.2, Section 6.12.11, Section 6.12.12, Section 6.13.11, and Section 6.13.12).

One of the key arguments in a call to a collective routine is a communicator that defines the group or groups of participating processes and provides a context for the operation. This is discussed further in Section 6.2. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. Thus, general datatypes are allowed and must match between sending and receiving processes as specified in Chapter 5. Several collective routines such as broadcast and gather have a single originating or receiving process. Such a process is called the *root*. Some arguments in the collective functions are specified as "significant only at root," and are ignored for all participants except the root. The reader is referred to Chapter 5 for information concerning communication buffers, general datatypes and type matching rules, and to Chapter 7 for information on how to define groups and create communicators.

The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory, see Section 5.1) between sender and receiver are still allowed.

27Collective operations can (but are not required to) complete as soon as the caller's 28participation in the collective communication is finished. A blocking operation is complete 29as soon as the call returns. A nonblocking (immediate) call requires a separate completion 30 call (cf. Section 3.7). The completion of a collective operation indicates that the caller is free  $^{31}$ to modify locations in the communication buffer. It does not indicate that other processes 32 in the group have completed or even started the operation (unless otherwise implied by the 33 description of the operation). Thus, a collective communication operation may, or may not, 34have the effect of synchronizing all participating MPI processes. 35

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. The collective operations do not have a message tag argument. A more detailed discussion of correct use of collective routines is found in Section 6.14.

- *Rationale.* The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI\_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.
- The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

<sup>48</sup> (End of rationale.)

2 3 4

5

6

7

8

9

10 11

12

13

14

15

16

17

18

19

20

21

41

42

43

44

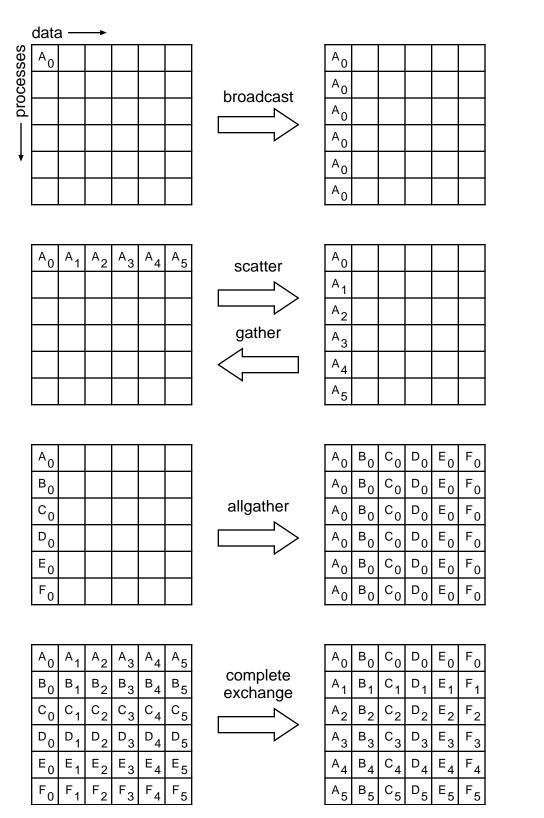


Figure 6.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data  $A_0$ , but after the broadcast all processes contain it.

 $1 \\ 2$ 

 $^{31}$ 

 $\frac{44}{45}$ 

Advice to users. It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.

On the other hand, a correct, portable program must allow for the fact that a collective call may be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 6.14. (End of advice to users.)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 6.14. (End of advice to implementors.)

17 18 19

20

21

Many of the descriptions of the collective routines provide illustrations in terms of blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are *not* correct MPI programs; for purposes of simplicity, they often assume infinite buffering.

2223 $^{24}$ 

25

#### 6.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating 26processes. The routines do not have group identifiers as explicit arguments. Instead, there 27is a communicator argument. Groups and communicators are discussed in full detail in 28Chapter 7. For the purposes of this chapter, it is sufficient to know that there are two types 29of communicators: intra-communicators and inter-communicators. An intra-communicator 30 can be thought of as an identifier for a single group of processes linked with a context. An  $^{31}$ inter-communicator identifies two distinct groups of processes linked with a context. 32

33 34

41

#### 6.2.1 Specifics for Intra-Communicator Collective Operations

35 All processes in the group identified by the intra-communicator must call the collective 36 routine. 37

In many cases, collective communication can occur "in place" for intra-communicators, 38 with the output buffer being identical to the input buffer. This is specified by providing 39 a special argument value, MPI\_IN\_PLACE, instead of the send buffer or the receive buffer 40 argument, depending on the operation performed.

42Rationale. The "in place" operations are provided to reduce unnecessary memory 43 motion by both the MPI implementation and by the user. Note that while the simple 44check of testing whether the send and receive buffers have the same address will 45work for some cases (e.g., MPI\_ALLREDUCE), they are inadequate in others (e.g., 46MPI\_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits 47 aliasing of arguments; the approach of using a special value to denote "in place" 48 operation eliminates that difficulty. (End of rationale.)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

Advice to users. By allowing the "in place" option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes INTENT must mark these as INOUT, not OUT.

Note that MPI\_IN\_PLACE is a special kind of value; it has the same restrictions on its use that MPI\_BOTTOM has (not usable in Fortran for initialization or assignment). See Section 2.5.4. (*End of advice to users.*)

## 6.2.2 Applying Collective Operations to Inter-Communicators

To understand how collective operations apply to inter-communicators, we can view most MPI intra-communicator collective operations as fitting one of the following categories (see, for instance, [63]):

All-To-All All processes contribute to the result. All processes receive the result.

•	MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHER_INIT,
	MPI_ALLGATHERV, MPI_IALLGATHERV, MPI_ALLGATHERV_INIT

- MPI\_ALLTOALL, MPI\_IALLTOALL, MPI\_ALLTOALL\_INIT, MPI\_ALLTOALLV, MPI\_IALLTOALLV, MPI\_ALLTOALLV\_INIT, MPI\_ALLTOALLW, MPI\_IALLTOALLW, MPI\_ALLTOALLW\_INIT
- MPI\_ALLREDUCE, MPI\_IALLREDUCE, MPI\_ALLREDUCE\_INIT, MPI\_REDUCE\_SCATTER\_BLOCK, MPI\_IREDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER\_BLOCK\_INIT, MPI\_REDUCE\_SCATTER, MPI\_IREDUCE\_SCATTER, MPI\_REDUCE\_SCATTER\_INIT
- All-To-One All processes contribute to the result. One process receives the result.
  - MPI\_GATHER, MPI\_IGATHER, MPI\_GATHER\_INIT, MPI\_GATHERV, MPI\_IGATHERV, MPI\_GATHERV\_INIT
  - MPI\_REDUCE, MPI\_IREDUCE, MPI\_REDUCE\_INIT,

MPI\_BARRIER, MPI\_IBARRIER, MPI\_BARRIER\_INIT

**One-To-All** One process contributes to the result. All processes receive the result.

MPI\_BCAST, MPI\_IBCAST, MPI\_BCAST\_INIT
MPI\_SCATTER, MPI\_ISCATTER, MPI\_SCATTER\_INIT, MPI\_SCATTERV, MPI\_ISCATTERV, MPI\_SCATTERV\_INIT

Other Collective operations that do not fit into one of the above categories.

• MPI\_SCAN, MPI\_ISCAN, MPI\_SCAN\_INIT MPI\_EXSCAN, MPI\_IEXSCAN, MPI\_EXSCAN\_INIT

The data movement patterns of MPI\_SCAN, MPI\_ISCAN, MPI\_EXSCAN, and MPI\_IEXSCAN do not fit this taxonomy.

The application of collective communication to inter-communicators is best described in terms of two groups. For example, an all-to-all MPI\_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 6.2). As another example, a one-to-all

 $41 \\ 42$ 

1	MPI_BCAST operation sends data from one member of one group to all members of the
2	other group. Collective computation operations such as MPI_REDUCE_SCATTER have a
3	similar interpretation (see Figure 6.3). For intra-communicators, these two groups are the
4	same. For inter-communicators, these two groups are distinct. For the all-to-all operations,
5	each such operation is described in two phases, so that it has a symmetric, full-duplex
6	behavior.
7	The following collective operations also apply to inter-communicators:
8 9	• MPI_BARRIER, MPI_IBARRIER, MPI_BARRIER_INIT,
10 11	• MPI_BCAST, MPI_IBCAST, MPI_BCAST_INIT,
12	<ul> <li>MPI_GATHER, MPI_IGATHER, MPI_GATHER_INIT, MPI_GATHERV,</li> </ul>
13	MPI_IGATHERV, MPI_GATHERV_INIT, MPI_GATHERV, MPI_GATHERV,
14	<ul> <li>MPI_SCATTER, MPI_ISCATTER, MPI_SCATTER_INIT, MPI_SCATTERV,</li> </ul>
15 16	MPI_ISCATTERV, MPI_SCATTERV_INIT,
17	• MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHER_INIT, MPI_ALLGATHERV,
18	MPI_IALLGATHERV, MPI_ALLGATHERV_INIT,
19	
20 21	• MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALL_INIT, MPI_ALLTOALLV,
21	MPI_IALLTOALLV, MPI_ALLTOALLV_INIT, MPI_ALLTOALLW, MPI_IALLTOALLW, MPI_ALLTOALLW_INIT,
23	WIFI_ALLI OALLW_INTT,
24	<ul> <li>MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_ALLREDUCE_INIT, MPI_REDUCE,</li> </ul>
25	MPI_IREDUCE, MPI_REDUCE_INIT,
26	<ul> <li>MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK,</li> </ul>
27	MPI_REDUCE_SCATTER_BLOCK_INIT, MPI_REDUCE_SCATTER,
28	MPI_IREDUCE_SCATTER, MPI_REDUCE_SCATTER_INIT.
29	
30 31	6.2.3 Specifics for Inter-Communicator Collective Operations
32	
33	All processes in both groups identified by the inter-communicator must call the collective
34	routine.
35	Note that the "in place" option for intra-communicators does not apply to inter-
36	communicators since in the inter-communicator case there is no communication from a process to itself.
37	For inter-communicator collective communication, if the operation is in the All-To-One
38 39	or One-To-All categories, then the transfer is unidirectional. The direction of the transfer is
39 40	indicated by a special value of the root argument. In this case, for the group containing the
41	root process, all processes in the group must call the routine using a special argument for
42	the root. For this, the root process uses the special root value MPI_ROOT; all other processes
	in the same group as the root use MPI PROC NULL. All processes in the other group (the

in the same group as the root use MPI\_PROC\_NULL. All processes in the other group (the group that is the remote group relative to the root process) must call the collective routine and provide the rank of the root. If the operation is in the All-To-All category, then the transfer is bidirectional.

- 47
- 48

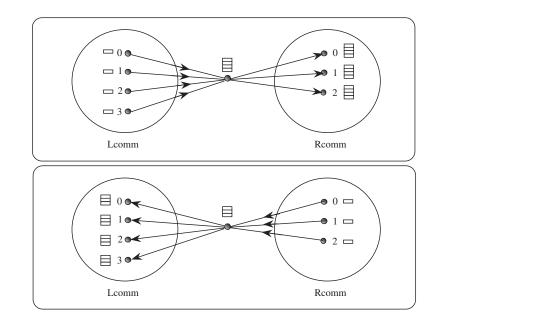


Figure 6.2: Inter-communicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

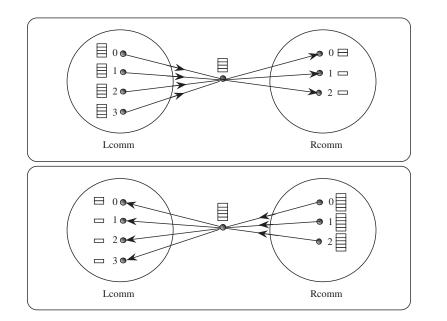


Figure 6.3: Inter-communicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

 $1 \\ 2$ 

 $14 \\ 15$ 

```
1
           Rationale. Operations in the All-To-One and One-To-All categories are unidirectional
\mathbf{2}
           by nature, and there is a clear way of specifying direction. Operations in the All-To-All
3
           category will often occur as part of an exchange, where it makes sense to communicate
4
           in both directions at once. (End of rationale.)
5
6
            Barrier Synchronization
     6.3
7
8
9
10
      MPI_BARRIER(comm)
11
        IN
                                               communicator (handle)
                  comm
12
13
      C binding
14
     int MPI_Barrier(MPI_Comm comm)
15
16
     Fortran 2008 binding
17
     MPI_Barrier(comm, ierror)
18
          TYPE(MPI_Comm), INTENT(IN) :: comm
19
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     Fortran binding
21
     MPI_BARRIER(COMM, IERROR)
22
          INTEGER COMM, IERROR
23
^{24}
          If comm is an intra-communicator, MPI_BARRIER blocks the caller until all group
25
     members have called it. The call returns at any process only after all group members have
26
      entered the call.
27
          If comm is an inter-communicator, MPI_BARRIER involves two groups. The call returns
28
      at processes in one group (group A) of the inter-communicator only after all members of
29
      the other group (group B) have entered the call (and vice versa). A process may return
30
      from the call before all processes in its own group have entered the call.
^{31}
32
            Broadcast
      6.4
33
34
35
36
      MPI_BCAST(buffer, count, datatype, root, comm)
37
        INOUT
                  buffer
                                               starting address of buffer (choice)
38
39
        IN
                  count
                                               number of entries in buffer (non-negative integer)
40
        IN
                  datatype
                                               datatype of buffer (handle)
41
        IN
                  root
                                               rank of broadcast root (integer)
42
        IN
                                               communicator (handle)
43
                  comm
44
45
      C binding
46
      int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
47
                     MPI_Comm comm)
48
```

CHAPTER 6. COLLECTIVE COMMUNICATION

int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype,	1
int root, MPI_Comm comm)	2
	3
Fortran 2008 binding	4
MPI_Bcast(buffer, count, datatype, root, comm, ierror)	5
TYPE(*), DIMENSION() :: buffer	6
INTEGER, INTENT(IN) :: count, root	7
TYPE(MPI_Datatype), INTENT(IN) :: datatype	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Bcast(buffer, count, datatype, root, comm, ierror) !(_c)	11
TYPE(*), DIMENSION() :: buffer	12
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	13
TYPE(MPI_Datatype), INTENT(IN) :: datatype	14
INTEGER, INTENT(IN) :: root	15
TYPE(MPI_Comm), INTENT(IN) :: comm	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
	18
Fortran binding	19
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)	20
<type> BUFFER(*)</type>	21
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR	22
If comm is an intra-communicator, MPI BCAST broadcasts a message from the process	23

If comm is an intra-communicator, MPI\_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group using the same arguments for comm and root. On return, the content of root's buffer is copied to all other processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI\_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful here.

If comm is an inter-communicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI\_ROOT in root. All other processes in group A pass the value MPI\_PROC\_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

## 6.4.1 Example using MPI\_BCAST

The examples in this section use intra-communicators.

Example 6.1 Broadcast 100 ints from process 0 to every process in the group.

MPI\_Comm comm;

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41 42

43

 $44 \\ 45$ 

 $46 \\ 47$ 

```
1
           int array[100];
\mathbf{2}
           int root=0;
3
           . . .
4
          MPI_Bcast(array, 100, MPI_INT, root, comm);
5
6
      As in many of our example code fragments, we assume that some of the variables (such as
\overline{7}
      comm in the above) have been assigned appropriate values.
8
9
      6.5
            Gather
10
11
12
13
      MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
14
       IN
                 sendbuf
                                              starting address of send buffer (choice)
15
16
       IN
                 sendcount
                                              number of elements in send buffer (non-negative
17
                                              integer)
18
       IN
                 sendtype
                                              datatype of send buffer elements (handle)
19
       OUT
                 recvbuf
                                              address of receive buffer (choice, significant only at
20
                                              root)
21
       IN
22
                  recvcount
                                               number of elements for any single receive
23
                                               (non-negative integer, significant only at root)
^{24}
       IN
                                               datatype of recv buffer elements (handle, significant
                 recvtype
25
                                               only at root)
26
                                              rank of receiving process (integer)
       IN
                  root
27
       IN
28
                                              communicator (handle)
                 comm
29
30
      C binding
31
      int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
32
                     void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
33
                     MPI_Comm comm)
34
      int MPI_Gather_c(const void *sendbuf, MPI_Count sendcount,
35
                     MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
36
                     MPI_Datatype recvtype, int root, MPI_Comm comm)
37
38
      Fortran 2008 binding
39
     MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
40
                     root, comm, ierror)
41
          TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
42
          INTEGER, INTENT(IN) :: sendcount, recvcount, root
43
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
44
          TYPE(*), DIMENSION(...) :: recvbuf
45
          TYPE(MPI_Comm), INTENT(IN) :: comm
46
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

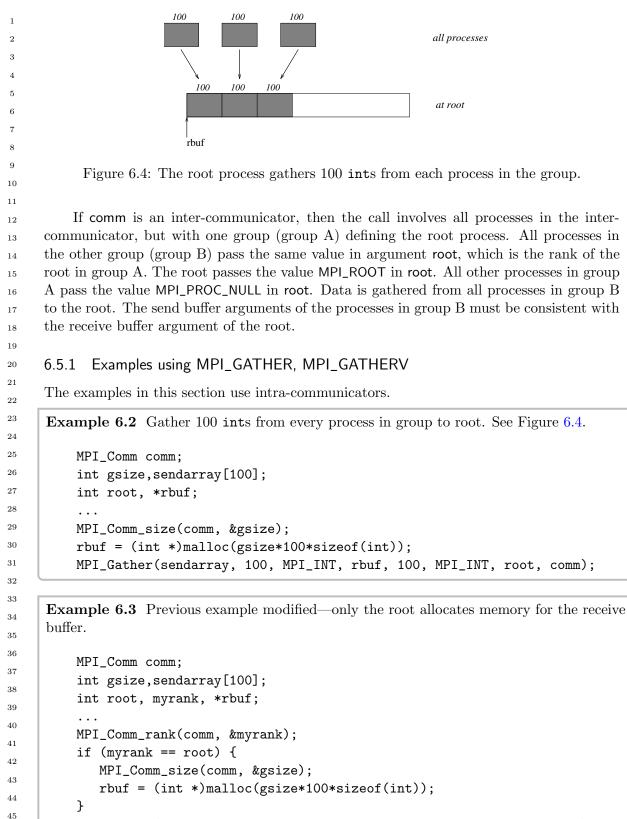
MPI\_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 1  $\mathbf{2}$ root, comm, ierror) !(\_c) 3 TYPE(\*), DIMENSION(...), INTENT(IN) :: sendbuf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 4 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 5 6 TYPE(\*), DIMENSION(..) :: recvbuf INTEGER, INTENT(IN) :: root 7 TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 10 Fortran binding 11 MPI\_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 12ROOT, COMM, IERROR) 13 <type> SENDBUF(\*), RECVBUF(\*) 14INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 1516If comm is an intra-communicator, each process (root process included) sends the con-17tents of its send buffer to the root process. The root process receives the messages and stores 18 them in rank order. The outcome is as if each of the n processes in the group (including 19 the root process) had executed a call to 20MPI\_Send(sendbuf, sendcount, sendtype, root, ...), 2122and the root had executed n calls to 23  $^{24}$ MPI\_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,...), 2526where extent(recvtype) is the type extent obtained from a call to MPI\_Type\_get\_extent. 27An alternative description is that the n messages sent by the processes in the group 28 are concatenated in rank order, and the resulting message is received by the root as if by a 29call to  $MPI_RECV$  (recvbuf, recvcount  $\cdot$  n, recvtype, ...). 30 The receive buffer is ignored for all non-root processes. 31General, derived datatypes are allowed for both sendtype and recvtype. The type signa-32 ture of sendcount, sendtype on each process must be equal to the type signature of recvcount, 33 recvtype at the root. This implies that the amount of data sent must be equal to the amount 34 of data received, pairwise between each process and the root. Distinct type maps between 35sender and receiver are still allowed. 36 All arguments to the function are significant on process root, while on other processes, 37 only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments 38 root and comm must have identical values on all processes. 39 The specification of counts and types should not cause any location on the root to be 40 written more than once. Such a call is erroneous. 41 Note that the recvcount argument at the root indicates the number of items it receives 42from *each* process, not the total number of items it receives. 43 The "in place" option for intra-communicators is specified by passing MPI\_IN\_PLACE 44as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and 45the contribution of the root to the gathered vector is assumed to be already in the correct 46place in the receive buffer. 4748 1 If comm is an inter-communicator, then the call involves all processes in the inter- $^{2}$ communicator, but with one group (group A) defining the root process. All processes in 3 the other group (group B) pass the same value in argument root, which is the rank of the 4 root in group A. The root passes the value MPI\_ROOT in root. All other processes in group  $\mathbf{5}$ A pass the value MPI\_PROC\_NULL in root. Data is gathered from all processes in group B 6 to the root. The send buffer arguments of the processes in group B must be consistent with  $\overline{7}$ the receive buffer argument of the root.

MPI\_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root,

11		comm)		
12	IN	sendbuf	starting address of send buffer (choice)	
13 14 15	IN	sendcount	number of elements in send buffer (non-negative integer)	
16	IN	sendtype	datatype of send buffer elements (handle)	
17 18	OUT	recvbuf	address of receive buffer (choice, significant only at root)	
19 20 21 22	IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)	
23 24 25 26	IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)	
27 28	IN	recvtype	datatype of recv buffer elements (handle, significant only at root)	
29 30	IN	root	rank of receiving process (integer)	
31	IN	comm	communicator (handle)	
32 33 34 35 36 37 38		Gatherv(const void *send void *recvbuf, cons MPI_Datatype recvty	<pre>buf, int sendcount, MPI_Datatype sendtype, st int recvcounts[], const int displs[], ype, int root, MPI_Comm comm) ndbuf, MPI_Count sendcount, ype woid *recybuf</pre>	
39 40 41		const MPI_Count rec	<pre>pe, void wiecebul, cvcounts[], const MPI_Aint displs[], vpe, int root, MPI_Comm comm)</pre>	
42	Fortran 2008 binding			
43 44	<pre>MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, ierror)</pre>			
45	TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf			
46	<pre>INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root</pre>			
47 48	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION() :: recvbuf			

8 9

```
1
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                             \mathbf{2}
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                             3
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
               recvtype, root, comm, ierror) !(_c)
                                                                                             5
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                             6
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
                                                                                             7
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                             8
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                             9
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                             10
    INTEGER, INTENT(IN) :: root
                                                                                             11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                             12
     INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                             13
                                                                                             14
Fortran binding
                                                                                             15
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
                                                                                             16
               RECVTYPE, ROOT, COMM, IERROR)
                                                                                             17
     <type> SENDBUF(*), RECVBUF(*)
                                                                                             18
     INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
                                                                                             19
                COMM, IERROR
                                                                                             20
    MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count
                                                                                            21
of data from each process, since recvcounts is now an array. It also allows more flexibility
                                                                                             22
as to where the data is placed on the root, by providing the new argument, displs.
                                                                                             23
    If comm is an intra-communicator, the outcome is as if each process, including the
                                                                                             24
root process, sends a message to the root,
                                                                                             25
                                                                                             26
   MPI_Send(sendbuf, sendcount, sendtype, root, ...),
                                                                                             27
                                                                                             28
and the root executes n receives,
                                                                                             29
                                                                                             30
   MPI_Recv(recvbuf+displs[i]· extent(recvtype), recvcounts[i], recvtype, i, ...).
                                                                                             31
    The data received from process j is placed into recvbuf of the root process beginning at
                                                                                             32
offset displs[i] elements (in terms of the recvtype).
                                                                                             33
    The receive buffer is ignored for all non-root processes.
                                                                                            34
    The type signature implied by sendcount, sendtype on process i must be equal to the
                                                                                            35
type signature implied by recvcounts[i], recvtype at the root. This implies that the amount
                                                                                            36
of data sent must be equal to the amount of data received, pairwise between each process
                                                                                            37
and the root. Distinct type maps between sender and receiver are still allowed, as illustrated
                                                                                             38
in Example 6.6.
                                                                                             39
    All arguments to the function are significant on process root, while on other processes,
                                                                                             40
                                                                                            41
only arguments sendbuf, sendcount, sendtype, root, and comm are significant. The arguments
root and comm must have identical values on all processes.
                                                                                             42
    The specification of counts, types, and displacements should not cause any location on
                                                                                             43
the root to be written more than once. Such a call is erroneous.
                                                                                             44
    The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE
                                                                                             45
as the value of sendbuf at the root. In such a case, sendcount and sendtype are ignored, and
                                                                                             46
the contribution of the root to the gathered vector is assumed to be already in the correct
                                                                                             47
                                                                                             48
place in the receive buffer.
```



MPI\_Gather(sendarray, 100, MPI\_INT, rbuf, 100, MPI\_INT, root, comm);

**Example 6.4** Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of gsize\*100 ints since type matching is defined pairwise between the root and each process in the gather.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size(comm, &gsize);
MPI_Type_contiguous(100, MPI_INT, &rtype);
MPI_Type_conmit(&rtype);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

**Example 6.5** Now have each process send 100 ints to root, but place each set (of 100) stride ints apart at receiving end. Use MPI\_GATHERV and the displs argument to achieve this effect. Assume *stride*  $\geq$  100. See Figure 6.5.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
    root, comm);
```

Note that the program is erroneous if stride < 100.

**Example 6.6** Same as Example 6.5 on the receiving side, but send the 100 ints from the 0th column of a  $100 \times 150$  int array, in C. See Figure 6.6.

```
MPI_Comm comm;
int gsize,sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs,i,*rcounts;
```

1

2

3

4 5

6

7

9

10

11

12

13

14

15 16

17

18

19 20

21

22

23

24 25

26 27

28

29

30

31

32

33

34

35

36

37 38

39 40

41

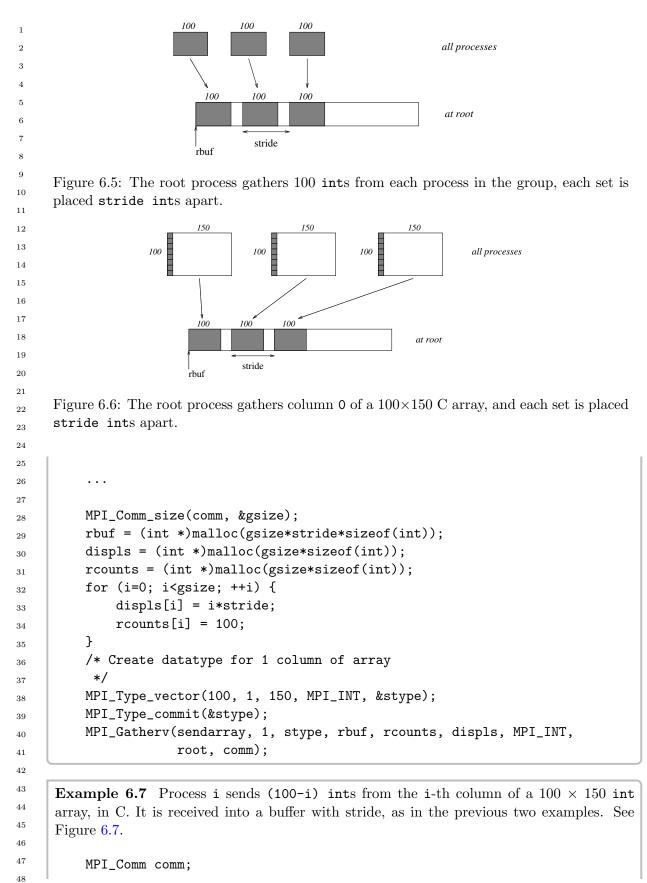
42 43

44

45

46

47



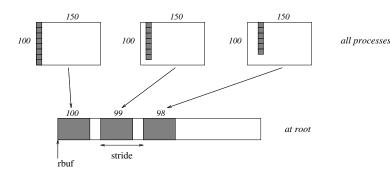


Figure 6.7: The root process gathers 100-i ints from column i of a  $100 \times 150$  C array, and each set is placed stride ints apart.

```
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts;
. . .
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = i*stride;
    rcounts[i] = 100-i;
                            /* note change from previous example */
}
/* Create datatype for the column we are sending
 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
/* sptr is the address of start of "myrank" column
 */
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

Note that a different amount of data is received from each process.

**Example 6.8** Same as Example 6.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that we read a column of a C array. A similar thing was done in Example 5.16, Section 5.1.14.

```
MPI_Comm comm;
int gsize, sendarray[100][150], *sptr;
```

1 2

3 4 5

6

9 10

11

12 13 14

15

16

17

18 19

20 21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40 41 42

43

44

 $45 \\ 46$ 

47

```
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs, i, *rcounts;
. . .
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = i*stride;
    rcounts[i] = 100-i;
}
/* Create datatype for one int, with extent of entire row
*/
MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &stype);
MPI_Type_commit(&stype);
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

**Example 6.9** Same as Example 6.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 6.8.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs,i,*rcounts,offset;
. . .
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
stride = (int *)malloc(gsize*sizeof(int));
. . .
/* stride[i] for i = 0 to gsize-1 is set somehow
*/
/* set up displs and rcounts vectors first
*/
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
```

1

 $\mathbf{2}$ 

3

4 5

6 7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23 24

25

26 27

28

29

30

31

32 33

34 35

36

37 38

39

40

41

42 43

44

45

46

47

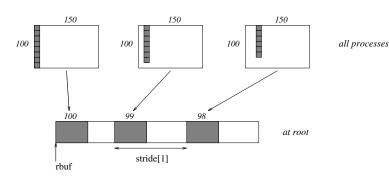


Figure 6.8: The root process gathers 100-i ints from column i of a  $100 \times 150$  C array, and each set is placed stride[i] ints apart (a varying stride).

```
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained
 */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
/* Create datatype for the column we are sending
 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

**Example 6.10** Process i sends num ints from the i-th column of a  $100 \times 150$  int array, in C. The complicating factor is that the various values of num are not known to root, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts,num;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
/* First, gather nums to root
```

48

1

 $\mathbf{2}$ 

3 4 5

6

9 10

11

12 13 14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

```
1
           */
\mathbf{2}
          rcounts = (int *)malloc(gsize*sizeof(int));
3
          MPI_Gather(&num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
4
          /* root now has correct rcounts, using these we set displs[] so
5
            * that data is placed contiguously (or concatenated) at receive end
6
           */
7
          displs = (int *)malloc(gsize*sizeof(int));
8
          displs[0] = 0;
9
          for (i=1; i<gsize; ++i) {</pre>
10
               displs[i] = displs[i-1]+rcounts[i-1];
11
          }
12
          /* And. create receive buffer
13
           */
14
          rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
15
                                                                             *sizeof(int));
16
          /* Create datatype for one int, with extent of entire row
17
           */
18
          MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &stype);
19
          MPI_Type_commit(&stype);
20
          sptr = &sendarray[0][myrank];
21
          MPI_Gatherv(sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
22
                        root, comm);
23
^{24}
     6.6
          Scatter
25
26
27
28
     MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
29
       IN
                 sendbuf
                                              address of send buffer (choice, significant only at
30
                                              root)
^{31}
       IN
32
                 sendcount
                                              number of elements sent to each process
33
                                              (non-negative integer, significant only at root)
34
       IN
                 sendtype
                                              datatype of send buffer elements (handle, significant
35
                                              only at root)
36
       OUT
                 recvbuf
                                              address of receive buffer (choice)
37
       IN
                                              number of elements in receive buffer (non-negative
38
                 recvcount
                                              integer)
39
40
       IN
                                              datatype of receive buffer elements (handle)
                 recvtype
41
       IN
                                              rank of sending process (integer)
                 root
42
       IN
                 comm
                                              communicator (handle)
43
44
45
     C binding
46
     int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
47
                     void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
48
                     MPI_Comm comm)
```

<pre>int MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,</pre>	1	
MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	2	
MPI_Datatype recvtype, int root, MPI_Comm comm)	3	
Fortran 2008 binding	4	
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	5	
root, comm, ierror)	6	
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	7	
INTEGER, INTENT(IN) :: sendcount, recvcount, root	8	
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	9	
TYPE(*), DIMENSION() :: recvbuf	10 11	
TYPE(MPI_Comm), INTENT(IN) :: comm	11	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12	
	14	
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	15	
root, comm, ierror) !(_c)	16	
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount	17	
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	18	
TYPE(*), DIMENSION() :: recvbuf	19	
INTEGER, INTENT(IN) :: root	20	
TYPE(MPI_Comm), INTENT(IN) :: comm	21	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22	
	23	
Fortran binding	24	
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	25	
ROOT, COMM, IERROR)	26	
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR</type>	27	
INTEGER SENDCOONT, SENDTIFE, RECVCOONT, RECVTIFE, ROOT, COMM, TERROR	28	
MPI_SCATTER is the inverse operation to MPI_GATHER.	29	
If $comm$ is an intra-communicator, the outcome is as if the root executed $n$ send	30	
operations,	31	
	32 33	
MPI_Send(sendbuf+i· sendcount· extent(sendtype), sendcount, sendtype, i,),	34	
and each process executed a receive,	35	
	36	
MPI_Recv(recvbuf, recvcount, recvtype, i,).	37	
	38	
An alternative description is that the root sends a message with MPI_Send(sendbuf,	39	
sendcount $n$ , sendtype,). This message is split into $n$ equal segments, the <i>i</i> -th segment is		
sent to the <i>i</i> -th process in the group, and each process receives this message as above.		
The send buffer is ignored for all non-root processes.	42	
The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type	43	
the type signature associated with recount, recyclype at all processes (nowever, the type		

The type signature associated with sendcount, sendtype at the root must be equal to the type signature associated with recvcount, recvtype at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

47 48

44

45

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

*Rationale.* Though not needed, the last restriction is imposed so as to achieve symmetry with MPI\_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (*End of rationale.*)

The "in place" option for intra-communicators is specified by passing MPI\_IN\_PLACE as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain n segments, where n is the group size; the *root*-th segment, which root should "send to itself," is not moved.

<sup>15</sup> If comm is an inter-communicator, then the call involves all processes in the inter-<sup>16</sup> communicator, but with one group (group A) defining the root process. All processes in <sup>17</sup> the other group (group B) pass the same value in argument root, which is the rank of the <sup>18</sup> root in group A. The root passes the value MPI\_ROOT in root. All other processes in group <sup>20</sup> A pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes <sup>21</sup> with the send buffer argument of the root.

23 24

25

MPI\_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

20		comm)	
26 27 28	IN	sendbuf	address of send buffer (choice, significant only at root)
29 30 31	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)
32 33 34 35	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (significant only at root)
36 37 38	IN	sendtype	datatype of send buffer elements (handle, significant only at root)
39	OUT	recvbuf	address of receive buffer (choice)
40 41	IN	recvcount	number of elements in receive buffer (non-negative integer)
42	IN	recvtype	datatype of receive buffer elements (handle)
43 44	IN	root	rank of sending process (integer)
45	IN	comm	communicator (handle)
46			
47	C bindir	ıg	
48	int MPI_	Scatterv(const voi	d *sendbuf, const int sendcounts[],

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

7

8

9 10

11

12

13

```
1
              const int displs[], MPI_Datatype sendtype, void *recvbuf,
                                                                                       2
              int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Scatterv_c(const void *sendbuf, const MPI_Count sendcounts[],
                                                                                       4
              const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
                                                                                       5
              MPI_Count recvcount, MPI_Datatype recvtype, int root,
                                                                                       6
              MPI Comm comm)
                                                                                       7
                                                                                       8
Fortran 2008 binding
                                                                                       9
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                      10
              recvtype, root, comm, ierror)
                                                                                      11
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
                                                                                      12
                                                                                      13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                      14
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                      15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      17
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
                                                                                      18
              recvtype, root, comm, ierror) !(_c)
                                                                                      19
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                      20
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount
                                                                                      21
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                      22
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                      23
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                      24
    INTEGER, INTENT(IN) :: root
                                                                                      25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      27
                                                                                      28
Fortran binding
                                                                                      29
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                      30
              RECVTYPE, ROOT, COMM, IERROR)
                                                                                      31
    <type> SENDBUF(*), RECVBUF(*)
                                                                                      32
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
                                                                                      33
               COMM, IERROR
                                                                                      34
    MPI_SCATTERV is the inverse operation to MPI_GATHERV.
                                                                                      35
    MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying
                                                                                      36
count of data to be sent to each process, since sendcounts is now an array. It also allows
                                                                                      37
more flexibility as to where the data is taken from on the root, by providing an additional
                                                                                      38
argument, displs.
                                                                                      39
    If comm is an intra-communicator, the outcome is as if the root executed n send oper-
                                                                                      40
ations.
                                                                                      41
                                                                                      42
   MPI_Send(sendbuf+displs[i] · extent(sendtype), sendcounts[i], sendtype, i,...),
                                                                                      43
                                                                                      44
and each process executed a receive,
                                                                                      45
                                                                                      46
   MPI_Recv(recvbuf, recvcount, recvtype, i,...).
                                                                                      47
                                                                                      48
    The send buffer is ignored for all non-root processes.
```

The type signature implied by sendcount[i], sendtype at the root must be equal to the type signature implied by recvcount, recvtype at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

<sup>6</sup> All arguments to the function are significant on process root, while on other processes,
 <sup>7</sup> only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments
 <sup>8</sup> root and comm must have identical values on all processes.

<sup>9</sup> The specification of counts, types, and displacements should not cause any location on <sup>10</sup> the root to be read more than once.

<sup>11</sup> The "in place" option for intra-communicators is specified by passing MPI\_IN\_PLACE <sup>12</sup> as the value of **recvbuf** at the root. In such a case, **recvcount** and **recvtype** are ignored, and <sup>13</sup> root "sends" no data to itself. The scattered vector is still assumed to contain n segments, <sup>14</sup> where n is the group size; the *root*-th segment, which root should "send to itself," is not <sup>15</sup> moved.

<sup>16</sup> If comm is an inter-communicator, then the call involves all processes in the inter-<sup>17</sup> communicator, but with one group (group A) defining the root process. All processes in <sup>18</sup> the other group (group B) pass the same value in argument root, which is the rank of the <sup>19</sup> root in group A. The root passes the value MPI\_ROOT in root. All other processes in group <sup>20</sup> A pass the value MPI\_PROC\_NULL in root. Data is scattered from the root to all processes <sup>21</sup> in group B. The receive buffer arguments of the processes in group B must be consistent <sup>22</sup> with the send buffer argument of the root.

2324

25

26 27

28

29 30

31

32

33

34

35

36

37

38 39

40

41

42 43

44

45

46 47

48

### 6.6.1 Examples using MPI\_SCATTER, MPI\_SCATTERV

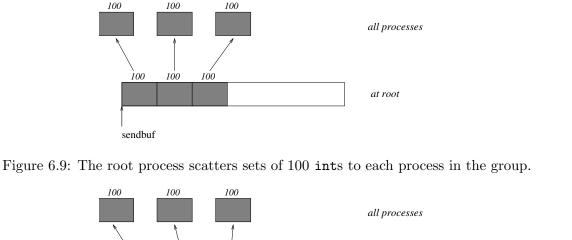
The examples in this section use intra-communicators.

**Example 6.11** The reverse of Example 6.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 6.9.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

**Example 6.12** The reverse of Example 6.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of MPI\_SCATTERV. Assume *stride*  $\geq$  100. See Figure 6.10.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
...
```



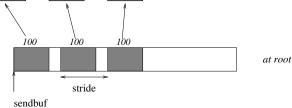


Figure 6.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

```
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
    root, comm);
```

**Example 6.13** The reverse of Example 6.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the *i*-th column of a  $100 \times 150$  C array. See Figure 6.11.

```
MPI_Comm comm;
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);
```

 $^{24}$ 

 $^{31}$ 

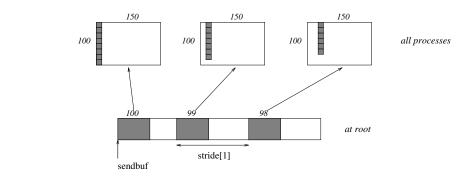


Figure 6.11: The root scatters blocks of 100-i ints into column i of a 100×150 C array. At the sending side, the blocks are stride[i] ints apart.

```
stride = (int *)malloc(gsize*sizeof(int));
. . .
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
. . .
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {</pre>
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving
 */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rptr = &recvarray[0][myrank];
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
             root, comm);
```

212

1

 $\mathbf{2}$ 

7

8

9

10

11

12 13 14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

 $^{31}$ 

32

33

34

## 6.7 Gather-to-all

MPI_AL	LGATHER(sendbuf, send	count, sendtype, recvbuf, recvcount, recvtype, comm)
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator (handle)
C bind	0	
int MPI	-	d *sendbuf, int sendcount,
	• •	sendtype, void *recvbuf, int recvcount,
	mer_Datatype i	recvtype, MPI_Comm comm)
int MPI	_Allgather_c(const v	oid *sendbuf, MPI_Count sendcount,
	• 1	endtype, void *recvbuf, MPI_Count recvcount,
	MPI_Datatype 1	recvtype, MPI_Comm comm)
Fortran	a 2008 binding	
	•	count, sendtype, recvbuf, recvcount, recvtype,
	comm, ierror)	
		INTENT(IN) :: sendbuf
	EGER, INTENT(IN) ::	-
		ENT(IN) :: sendtype, recvtype
	<pre>PE(*), DIMENSION()</pre>	
	PE(MPI_Comm), INTENT(	
TN.I	EGER, OPTIONAL, INTE	NT(UUT) :: lerror
MPI_All	gather(sendbuf, send	count, sendtype, recvbuf, recvcount, recvtype,
	comm, ierror)	!(_c)
		INTENT(IN) :: sendbuf
		KIND), INTENT(IN) :: sendcount, recvcount
	• 1	ENT(IN) :: sendtype, recvtype
	E(*), DIMENSION()	
	E(MPI_Comm), INTENT(	
TNJ	EGER, OPTIONAL, INTE	NI(UUI) :: lerror
Fortran	ı binding	
MPI_ALL	GATHER(SENDBUF, SEND	COUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
	COMM, IERROR)	
v	<pre>pe&gt; SENDBUF(*), RECV</pre>	
INT	EGER SENDCOUNT, SEND	TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

1	MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive
2	the result, instead of just the root. The block of data sent from the j-th process is received
3	by every process and placed in the j-th block of the buffer recvbuf.
4	The type signature associated with sendcount, sendtype, at a process must be equal to
5	the type signature associated with recvcount, recvtype at any other process.
6	If comm is an intra-communicator, the outcome of a call to MPI_ALLGATHER() is
7	as if all processes executed n calls to
8	
9	MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
10	recvtype,root,comm)
11	
12	for root = 0,, n-1. The rules for correct usage of MPI_ALLGATHER are easily found
13	from the corresponding rules for MPI_GATHER.
14	The "in place" option for intra-communicators is specified by passing the value
15	MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored.
16	Then the input data of each process is assumed to be in the area where that process would
17	receive its own contribution to the receive buffer.
18	If comm is an inter-communicator, then each process of one group (group A) contributes
19	sendcount data items; these data are concatenated and the result is stored at each process
20	in the other group (group B). Conversely the concatenation of the contributions of the
21	processes in group B is stored at each process in group A. The send buffer arguments in
22	group A must be consistent with the receive buffer arguments in group B, and vice versa.
23	
24	Advice to users. The communication pattern of MPI_ALLGATHER executed on an
25	intercommunication domain need not be symmetric. The number of items sent by
26	processes in group A (as specified by the arguments sendcount, sendtype in group A
27	and the arguments recvcount, recvtype in group B), need not equal the number of
28	items sent by processes in group B (as specified by the arguments sendcount, sendtype
29	in group B and the arguments recvcount, recvtype in group A). In particular, one can
30	move data in only one direction by specifying sendcount = 0 for the communication in the neuronal direction $(Find ef e drive te mean)$
31	in the reverse direction. (End of advice to users.)
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

MPI_A	LLGATHERV(sendbuf, se comm)	ndcount, sendtype, recvbuf, recvcounts, displs, recvtype,	1 2
IN	sendbuf	starting address of send buffer (choice)	3 4
IN	sendcount	number of elements in send buffer (non-negative integer)	5 6
IN	sendtype	datatype of send buffer elements (handle)	7
OUT	recvbuf	address of receive buffer (choice)	8 9
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	10 11 12
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	13 14 15
IN	recvtype	datatype of receive buffer elements (handle)	16 17
IN	comm	communicator (handle)	18
	I_Allgatherv(const v MPI_Datatype const int dis	oid *sendbuf, int sendcount, sendtype, void *recvbuf, const int recvcounts[], spls[], MPI_Datatype recvtype, MPI_Comm comm)	20 21 22 23 24
int MP	MPI_Datatype const MPI_Cou	<pre>void *sendbuf, MPI_Count sendcount, sendtype, void *recvbuf, unt recvcounts[], const MPI_Aint displs[], recvtype, MPI_Comm comm)</pre>	25 26 27 28
MPI_Al TY IN TY TY TY	recvtype, com PE(*), DIMENSION() TEGER, INTENT(IN) ::	<pre>, INTENT(IN) :: sendbuf sendcount, recvcounts(*), displs(*) TENT(IN) :: sendtype, recvtype</pre>	29 30 31 32 33 34 35 36 37 28
<pre>MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,</pre>			38 39 40 41 42 43 44 45 46 47

1	Fortran binding
2	MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
3	RECVTYPE, COMM, IERROR)
4	<type> SENDBUF(*), RECVBUF(*)</type>
5	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
6	IERROR
7	
8	MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes re-
9	ceive the result, instead of just the root. The block of data sent from the j-th process is
10	received by every process and placed in the j-th block of the buffer recvbuf. These blocks
11	need not all be the same size.
12	The type signature associated with sendcount, sendtype, at process j must be equal to
	the type signature associated with recvcounts[j], recvtype at any other process.
13	If comm is an intra-communicator, the outcome is as if all processes executed calls to
14	
15	<pre>MPI_Gatherv(sendbuf,sendcount,sendtype,recvbuf,recvcounts,displs,</pre>
16	<pre>recvtype,root,comm),</pre>
17	
18	for root = 0,, n-1. The rules for correct usage of MPI_ALLGATHERV are easily
19	found from the corresponding rules for MPI_GATHERV.
20	The "in place" option for intra-communicators is specified by passing the value
21	MPI_IN_PLACE to the argument sendbuf at all processes. In such a case, sendcount and
22	sendtype are ignored, and the input data of each process is assumed to be in the area where
23	that process would receive its own contribution to the receive buffer.
24	If comm is an inter-communicator, then each process of one group (group A) contributes
25	sendcount data items; these data are concatenated and the result is stored at each process
26	in the other group (group B). Conversely the concatenation of the contributions of the
27	processes in group B is stored at each process in group A. The send buffer arguments in
28	group A must be consistent with the receive buffer arguments in group B, and vice versa.
29	
30	6.7.1 Example using MPI_ALLGATHER
31	The example in this section uses intra-communicators.
32	
33	<b>Example 6.14</b> The all-gather version of Example 6.2. Using MPI_ALLGATHER, we will
34	gather 100 ints from every process in the group to every process.
35	
36	MPI_Comm comm;
37	<pre>int gsize,sendarray[100];</pre>
38	<pre>int *rbuf;</pre>
39	
40	<pre>MPI_Comm_size(comm, &amp;gsize);</pre>
41	<pre>rbuf = (int *)malloc(gsize*100*sizeof(int));</pre>
42	<pre>MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);</pre>
43	
44	After the call, every process has the group-wide concatenation of the sets of data.
45	
46	
47	
48	

# 6.8 All-to-All Scatter/Gather

			2
			3 4
MPI_Al	LTOALL(sendbuf, sendcou	unt, sendtype, recvbuf, recvcount, recvtype, comm)	5
IN	sendbuf	starting address of send buffer (choice)	6
IN	sendcount	number of elements sent to each process (non-negative integer)	7 8
IN	sendtype	datatype of send buffer elements (handle)	9 10
OUT	recvbuf	address of receive buffer (choice)	11
IN	recvcount	number of elements received from any process (non-negative integer)	12 13
IN	recvtype	datatype of receive buffer elements (handle)	14
IN	comm	communicator (handle)	15 16
			17
C bind	ling		18
int MP		*sendbuf, int sendcount, MPI_Datatype sendtype,	19
		<pre>int recvcount, MPI_Datatype recvtype,</pre>	20 21
	MPI_Comm comm)		21
int MP		id *sendbuf, MPI_Count sendcount,	23
		endtype, void *recvbuf, MPI_Count recvcount,	24
	MPI_Datatype r	ecvtype, MPI_Comm comm)	25
Fortra	n 2008 binding		26
	_	ount, sendtype, recvbuf, recvcount, recvtype,	27 28
TY	-	INTENT(IN) :: sendbuf	29
	TEGER, INTENT(IN) :: :		30
		ENT(IN) :: sendtype, recvtype	31
	PE(*), DIMENSION()		32
TY	PE(MPI_Comm), INTENT()	IN) :: comm	33
IN	TEGER, OPTIONAL, INTE	NT(OUT) :: ierror	34
ΜΡΤ ΔΙ	ltoall(sendbuf sendc	ount, sendtype, recvbuf, recvcount, recvtype,	35
	comm, ierror)		36
TY	-	INTENT(IN) :: sendbuf	37
		KIND), INTENT(IN) :: sendcount, recvcount	38
		ENT(IN) :: sendtype, recvtype	39
	PE(*), DIMENSION()		40
	PE(MPI_Comm), INTENT(		41
	TEGER, OPTIONAL, INTE		42 43
Fortro	n binding		44
		OUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	45
· III I _ ALL	COMM, IERROR)	50.1, 5E.75111E, 16504501, 1650400001, 16504111E,	46
<t.< td=""><td>ype&gt; SENDBUF(*), RECV</td><td>BUF(*)</td><td>47</td></t.<>	ype> SENDBUF(*), RECV	BUF(*)	47
	· -	TYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	48

1 2	MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received
3	by process j and is placed in the i-th block of recvbuf.
4	The type signature associated with sendcount, sendtype, at a process must be equal to
5	the type signature associated with recvcount, recvtype at any other process. This implies
6	that the amount of data sent must be equal to the amount of data received, pairwise between
7	every pair of processes. As usual, however, the type maps may be different.
8	If comm is an intra-communicator, the outcome is as if each process executed a send
9 10	to each process (itself included) with a call to,
10	MPI_Send(sendbuf+i· sendcount· extent(sendtype),sendcount,sendtype,i,),
12	
13	and a receive from every other process with a call to,
14 15	$MPI\_Recv(recvbuf+i\cdot recvcount\cdot extent(recvtype), recvcount, recvtype, i, \ldots).$
16	All arguments on all processes are significant. The argument <b>comm</b> must have identical
17	values on all processes.
18 19	The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE to
20	the argument sendbuf at $all$ processes. In such a case, sendcount and sendtype are ignored.
20	The data to be sent is taken from the <b>recvbuf</b> and replaced by the received data. Data sent
22	and received must have the same type map as specified by <b>recvcount</b> and <b>recvtype</b> .
23	Detionale For lange MDI ALLTOALL instances allegating both and and maring
24	<i>Rationale.</i> For large MPI_ALLTOALL instances, allocating both send and receive buffers may consume too much memory. The "in place" option effectively halves the
25	application memory consumption and is useful in situations where the data to be sent
26	will not be used by the sending process after the MPI_ALLTOALL exchange (e.g., in
27	parallel Fast Fourier Transforms). (End of rationale.)
28	Farance - and - care -
29	Advice to implementors. Users may opt to use the "in place" option in order to
30	conserve memory. Quality MPI implementations should thus strive to minimize system
31	buffering. (End of advice to implementors.)
32	
33 34	If comm is an inter-communicator, then the outcome is as if each process in group A
35	sends a message to each process in group B, and vice versa. The j-th send buffer of process
36	i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.
37	
38	Advice to users. When a complete exchange is executed on an intercommunication
39	domain, then the number of data items sent from processes in group A to processes
40	in group B need not equal the number of items sent in the reverse direction. In
41	particular, one can have unidirectional communication by specifying $sendcount = 0$ in
42	the reverse direction. (End of advice to users.)
43	
44	
45	
46	
47	
48	

MPI_ALLT	OALLV(sendbuf, sendcounts, s recvtype, comm)	displs, sendtype, recvbuf, recvcounts, rdispls,	1 $2$
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank	4 5 6 7
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j	8 9 10
IN	sendtype	datatype of send buffer elements (handle)	11 12
OUT	recvbuf	address of receive buffer (choice)	12
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank	14 15 16
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	17 18 19 20
IN	recvtype	datatype of receive buffer elements (handle)	21
IN	comm	communicator (handle)	22
	<pre>Iltoallv(const void *send const int sdispls[], const int recvcounts MPI_Datatype recvtyp</pre>		24 25 26 27 28 29 30
<pre>int MPI_Alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>			30 31 32 33 34
MPI_Allto TYPE( INTEG TYPE(	<pre>rdispls, recvtype, c *), DIMENSION(), INTEN</pre>	<pre>I(IN) :: sendbuf unts(*), sdispls(*), recvcounts(*), ) :: sendtype, recvtype</pre>	35 36 37 38 39 40 41 42
TYPE( INTEG	<pre>MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT)</pre>	Comm	$43 \\ 44 \\ 45 \\ 46$

```
1
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
2
                      recvcounts(*)
3
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
          TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
5
          TYPE(*), DIMENSION(..) :: recvbuf
6
          TYPE(MPI_Comm), INTENT(IN) :: comm
7
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
10
                     RDISPLS, RECVTYPE, COMM, IERROR)
11
          <type> SENDBUF(*), RECVBUF(*)
12
          INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
13
                      RECVTYPE, COMM, IERROR
14
15
          MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for
16
      the send is specified by sdispls and the location of the placement of the data on the receive
17
     side is specified by rdispls.
18
          If comm is an intra-communicator, then the j-th block sent from process i is received
19
      by process j and is placed in the i-th block of recvbuf. These blocks need not all have the
20
      same size.
21
          The type signature associated with sendcounts[i], sendtype at process i must be equal
22
      to the type signature associated with recvcounts[i], recvtype at process j. This implies that
23
      the amount of data sent must be equal to the amount of data received, pairwise between
^{24}
      every pair of processes. Distinct type maps between sender and receiver are still allowed.
25
          The outcome is as if each process sent a message to every other process with,
26
         MPI_Send(sendbuf+sdispls[i] \cdot extent(sendtype), sendcounts[i], sendtype, i, ...),
27
28
      and received a message from every other process with a call to
29
30
         MPI_Recv(recvbuf+rdispls[i] extent(recvtype),recvcounts[i],recvtype,i,...).
^{31}
32
          All arguments on all processes are significant. The argument comm must have identical
33
      values on all processes.
34
          The "in place" option for intra-communicators is specified by passing MPI_IN_PLACE to
35
      the argument sendbuf at all processes. In such a case, sendcounts, sdispls and sendtype are
36
      ignored. The data to be sent is taken from the recvbuf and replaced by the received data.
37
      Data sent and received must have the same type map as specified by the recvcounts array
38
      and the recvtype, and is taken from the locations of the receive buffer specified by rdispls.
39
40
                                Specifying the "in place" option (which must be given on all
           Advice to users.
41
           processes) implies that the same amount and type of data is sent and received between
42
           any two processes in the group of the communicator. Different pairs of processes can
43
           exchange different amounts of data. Users must ensure that recvcounts[j] and recvtype
44
           on process i match recvcounts[i] and recvtype on process j. This symmetric exchange
45
           can be useful in applications where the data to be sent will not be used by the sending
46
           process after the MPI_ALLTOALLV exchange. (End of advice to users.)
47
```

If comm is an inter-communicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

*Rationale.* The definitions of MPI\_ALLTOALL and MPI\_ALLTOALLV give as much flexibility as one would achieve by specifying n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (*End of rationale.*)

Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (End of advice to implementors.)

	recvtypes, comm)		19
IN	sendbuf	starting address of send buffer (choice)	20
IN	sendcounts	non-negative integer array (of length group size)	21
		specifying the number of elements to send to each	22
		rank	23
IN	sdispls	integer array (of length group size). Entry j specifies	24
		the displacement in bytes (relative to sendbuf) from	25
		which to take the outgoing data destined for process	26
		j (array of integers)	27 28
IN	sendtypes	array of datatypes (of length group size). Entry j	28 29
	51	specifies the type of data to send to process j (array	30
		of handles)	31
OUT	recvbuf	address of receive buffer (choice)	32
IN	recvcounts	non-negative integer array (of length group size)	33
		specifying the number of elements that can be	34
		received from each rank	35
IN	rdispls	integer array (of length group size). Entry i specifies	36
		the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i	37
			38
		(array of integers)	39
IN	recvtypes	array of datatypes (of length group size). Entry i	40 41
		specifies the type of data received from process i	41
		(array of handles)	43
IN	comm	communicator (handle)	44
		×	45
C bindi	ng		46
	0	endbuf, const int sendcounts[],	47

MPI\_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

  $\mathbf{2}$ 

```
1
                    void *recvbuf, const int recvcounts[], const int rdispls[],
\mathbf{2}
                    const MPI_Datatype recvtypes[], MPI_Comm comm)
3
     int MPI_Alltoallw_c(const void *sendbuf, const MPI_Count sendcounts[],
4
                    const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
5
                    void *recvbuf, const MPI_Count recvcounts[],
6
                    const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
7
                    MPI_Comm comm)
8
9
     Fortran 2008 binding
10
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
11
                    rdispls, recvtypes, comm, ierror)
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
13
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
14
                     rdispls(*)
15
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
16
         TYPE(*), DIMENSION(...) :: recvbuf
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
20
                    rdispls, recvtypes, comm, ierror) !(_c)
21
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
22
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
23
                     recvcounts(*)
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
25
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
26
         TYPE(*), DIMENSION(...) :: recvbuf
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     Fortran binding
31
     MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
32
                    RDISPLS, RECVTYPES, COMM, IERROR)
33
         <type> SENDBUF(*), RECVBUF(*)
34
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
35
                     RDISPLS(*), RECVTYPES(*), COMM, IERROR
36
         MPI_ALLTOALLW is the most general form of complete exchange. Like
37
     MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-
38
     lows separate specification of count, displacement and datatype. In addition, to allow max-
39
     imum flexibility, the displacement of blocks within the send and receive buffers is specified
40
     in bytes.
41
         If comm is an intra-communicator, then the j-th block sent from process i is received
42
     by process j and is placed in the i-th block of recvbuf. These blocks need not all have the
43
     same size.
44
         The type signature associated with sendcounts[i], sendtypes[i] at process i must be equal
45
     to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that
46
     the amount of data sent must be equal to the amount of data received, pairwise between
47
     every pair of processes. Distinct type maps between sender and receiver are still allowed.
48
```

The outcome is as if each process sent a message to every other process with	1
	2
MPI_Send(sendbuf+sdispls[i],sendcounts[i],sendtypes[i] ,i,),	3
	4
and received a message from every other process with a call to	5
	6
MPI_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i] ,i,).	7

All arguments on all processes are significant. The argument comm must describe the same communicator on all processes.

Like for MPI\_ALLTOALLV, the "in place" option for intra-communicators is specified by passing MPI\_IN\_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the received must and receives arrays, and is taken from the locations of the receive buffer specified by rdispls.

If **comm** is an inter-communicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

*Rationale.* The MPI\_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have sendcounts[i] = 0, this achieves an MPI\_SCATTERW function. (*End of rationale.*)

#### 6.9 Global Reduction Operations

The functions in this section perform a global reduce operation (for example sum, maximum, and logical and) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

```
224
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.9.1 Reduce
\mathbf{2}
3
4
     MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
5
       IN
                sendbuf
                                            address of send buffer (choice)
6
       OUT
7
                recvbuf
                                            address of receive buffer (choice, significant only at
8
                                            root)
9
       IN
                count
                                            number of elements in send buffer (non-negative
10
                                            integer)
11
       IN
                datatype
                                            datatype of elements of send buffer (handle)
12
       IN
                                            reduce operation (handle)
13
                ор
14
       IN
                root
                                            rank of root process (integer)
15
       IN
                                            communicator (handle)
                comm
16
17
     C binding
18
     int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
19
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
20
21
     int MPI_Reduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,
22
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
23
     Fortran 2008 binding
^{24}
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
25
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
26
         TYPE(*), DIMENSION(..) :: recvbuf
27
         INTEGER, INTENT(IN) :: count, root
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Op), INTENT(IN) :: op
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror) !(_c)
34
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
35
         TYPE(*), DIMENSION(..) :: recvbuf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Op), INTENT(IN) :: op
39
         INTEGER, INTENT(IN) :: root
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     Fortran binding
43
     MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
44
          <type> SENDBUF(*), RECVBUF(*)
45
         INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
46
47
48
```

If comm is an intra-communicator, MPI\_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the 3 combined value in the output buffer of the process with rank root. The input buffer is 4 defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for 6  $\overline{7}$ count, datatype, op, root and comm. Thus, all processes provide input buffers of the same length, with elements of the same type as the output buffer at the root. Each process 9 can provide one element, or a sequence of elements, in which case the combine operation 10 is executed element-wise on each entry of the sequence. For example, if the operation 11is MPI\_MAX and the send buffer contains two elements that are floating point numbers 12(count = 2 and datatype = MPI\_FLOAT), then recvbuf(1) =  $global \max(sendbuf(1))$  and 13 recvbuf(2) = global max(sendbuf(2)).

Section 6.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 6.9.5.

The operation **op** is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that MPI\_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of ranks. (End of advice to *implementors.*)

Advice to users. Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 6.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with MPI\_GATHER), applying the reduction operation in the desired order (e.g., with MPI\_REDUCE\_LOCAL), and if needed, broadcast or scatter the result to the other processes (e.g., with MPI\_BCAST). (End of advice to users.)

The datatype argument of MPI\_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 6.9.2 and Section 6.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI\_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described

1  $\mathbf{2}$ 

 $\mathbf{5}$ 

8

1415

16

17

18

1920

21

22

23 $^{24}$ 

25

26

27

28

29

30 31

32

33

34

35

36

37

38

39

40 41

42

43

44

45

46

47

by such a datatype, which may contain several basic values. This is further explained in  $\mathbf{2}$ Section 6.9.5. 3

> Users should make no assumptions about how MPI\_REDUCE is Advice to users. implemented. It is safest to ensure that the same function is passed to MPI\_REDUCE by each process. (End of advice to users.)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

The "in place" option for intra-communicators is specified by passing the value MPI\_IN\_PLACE to the argument sendbuf at the root. In such a case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an inter-communicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI\_ROOT in root. All other processes in group A pass the value MPI\_PROC\_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

19Predefined Reduction Operations 6.9.2 20

21The following predefined operations are supplied for MPI\_REDUCE and related functions 22MPI\_ALLREDUCE, MPI\_REDUCE\_SCATTER\_BLOCK, MPI\_REDUCE\_SCATTER,

23MPI\_SCAN, MPI\_EXSCAN, all nonblocking variants of those (see Section 6.12), and

 $^{24}$ MPI\_REDUCE\_LOCAL. These operations are invoked by placing the following in op.

2526

Name Meaning 2728MPI\_MAX maximum 29 MPI\_MIN minimum 30 MPI\_SUM  $\operatorname{sum}$  $^{31}$ MPI\_PROD product 32 logical and MPI\_LAND 33 bit-wise and MPI\_BAND 34 logical or MPI\_LOR 35 MPI\_BOR bit-wise or 36 MPI\_LXOR logical exclusive or (xor) 37 MPI\_BXOR bit-wise exclusive or (xor) 38 MPI\_MAXLOC max value and location 39 MPI\_MINLOC min value and location 40

The two operations MPI\_MINLOC and MPI\_MAXLOC are discussed separately in Sec-41 tion 6.9.4. For the other predefined operations, we enumerate below the allowed combi-42nations of op and datatype arguments. First, define groups of MPI basic datatypes in the 43 following way. 44

45

46

C integer: 47

48

MPI\_INT, MPI\_LONG, MPI\_SHORT, MPI\_UNSIGNED\_SHORT, MPI\_UNSIGNED,

1

4

5

6 7

8

9

10

11

12

13

14

15

16

17

	MPI_UNSIGNED_LONG,	1
	MPI_LONG_LONG_INT,	2
	MPI_LONG_LONG (as synonym),	3
	MPI_UNSIGNED_LONG_LONG,	4
	MPI_SIGNED_CHAR,	5
	MPI_UNSIGNED_CHAR,	6
	MPI_INT8_T, MPI_INT16_T,	7
	MPI_INT32_T, MPI_INT64_T,	8
	MPI_UINT8_T, MPI_UINT16_T,	9
	MPI_UINT32_T, and MPI_UINT64_T	10
Fortran integer:	MPI_INTEGER	11
5	and handles returned from	12
	MPI_TYPE_CREATE_F90_INTEGER	13
	and, if available, MPI_INTEGER1,	13
	MPI_INTEGER2, MPI_INTEGER4,	
	MPI_INTEGER8, and MPI_INTEGER16	15
Floating point:	MPI_FLOAT, MPI_DOUBLE, MPI_REAL,	16
	MPI_DOUBLE_PRECISION,	17
	MPI_LONG_DOUBLE,	18
	and handles returned from	19
	MPI_TYPE_CREATE_F90_REAL	20
	and, if available, MPI_REAL2,	21
	MPI_REAL4, MPI_REAL8, and MPI_REAL16	22
Logical:	MPI_LOGICAL, MPI_C_BOOL,	23
0	and MPI_CXX_BOOL	24
Complex:	MPI_COMPLEX, MPI_C_COMPLEX,	25
-	MPI_C_FLOAT_COMPLEX (as synonym),	26
	MPI_C_DOUBLE_COMPLEX,	27
	MPI_C_LONG_DOUBLE_COMPLEX,	28
	MPI_CXX_FLOAT_COMPLEX,	29
	MPI_CXX_DOUBLE_COMPLEX,	30
	MPI_CXX_LONG_DOUBLE_COMPLEX,	31
	and handles returned from	32
	MPI_TYPE_CREATE_F90_COMPLEX	33
	and, if available, MPI_DOUBLE_COMPLEX,	34
	MPI_COMPLEX4, MPI_COMPLEX8,	35
	MPI_COMPLEX16, and MPI_COMPLEX32	36
Byte:	MPI_BYTE	37
Multi-language types:	MPI_AINT, MPI_OFFSET, and MPI_COUNT	38
Norre the collid detectors of four or de		39
Now, the valid datatypes for each	operation are specified below.	40
		41
Om	Allowed Types	42
Ор	Allowed Types	
	Cinterna Fratura interna Flastica asiat	43
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point,	44
MDI SIIM MDI DDOD	Multi-language types	45
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex, Multi-language types	46
MRITAND MRITOP MRITYOP	0 0 0 1	47
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical	48

```
1
       MPI_BAND, MPI_BOR, MPI_BXOR
                                             C integer, Fortran integer, Byte, Multi-language types
\mathbf{2}
          These operations together with all listed datatypes are valid in all supported program-
3
     ming languages, see also Reduce Operations on page 846 in Section 19.3.6.
4
          The following examples use intra-communicators.
5
6
     Example 6.15 A routine that computes the dot product of two vectors that are distributed
7
     across a group of processes and returns the answer at node zero.
8
9
     SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
10
     REAL a(m), b(m)
                               ! local slice of array
11
     REAL c
                               ! result (at node zero)
12
     REAL sum
     INTEGER m, comm, i, ierr
13
14
     ! local sum
15
16
     sum = 0.0
17
     DO i = 1, m
18
         sum = sum + a(i)*b(i)
19
     END DO
20
21
      ! global sum
^{22}
     CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
23
     RETURN
^{24}
     END
25
26
     Example 6.16 A routine that computes the product of a vector and an array that are
27
     distributed across a group of processes and returns the answer at node zero.
28
29
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
30
     REAL a(m), b(m,n)
                             ! local slice of array
31
     REAL c(n)
                              ! result
32
     REAL sum(n)
33
     INTEGER m, n, comm, i, j, ierr
34
35
      ! local sum
36
     DO j=1,n
37
         sum(j) = 0.0
38
         DO i=1,m
39
            sum(j) = sum(j) + a(i)*b(i,j)
40
         END DO
41
     END DO
42
43
      ! global sum
44
     CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
45
46
      ! return result at node zero (and garbage at the other nodes)
47
     RETURN
48
```

CHAPTER 6. COLLECTIVE COMMUNICATION

#### END

#### 6.9.3 Signed Characters and Reductions

The types MPI\_SIGNED\_CHAR and MPI\_UNSIGNED\_CHAR can be used in reduction operations. MPI\_CHAR, MPI\_WCHAR, and MPI\_CHARACTER (which represent printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI\_CHAR, MPI\_WCHAR, and MPI\_CHARACTER will be translated so as to preserve the printable character, whereas MPI\_SIGNED\_CHAR and MPI\_UNSIGNED\_CHAR will be translated so as to preserve the integer value.

Advice to users. The types MPI\_CHAR, MPI\_WCHAR, and MPI\_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI\_SIGNED\_CHAR and MPI\_UNSIGNED\_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

#### 6.9.4 MINLOC and MAXLOC

The operator MPI\_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI\_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI\_MAXLOC is:

$$\left(\begin{array}{c} u\\i\end{array}\right)\circ\left(\begin{array}{c} v\\j\end{array}\right)=\left(\begin{array}{c} w\\k\end{array}\right)$$

where

$$w = \max(u, v)$$

and

$$k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

MPI\_MINLOC is defined similarly:

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ k \end{pmatrix}$$
<sup>38</sup>
<sup>39</sup>
<sup>40</sup>
<sup>41</sup>

where

$$w = \min(u, v)$$
<sup>43</sup>

and

$$\left( \begin{array}{ccc} i & \text{if } u < v \end{array} \right)^{46}$$

$$k = \begin{cases} \min(i,j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$
<sup>47</sup>

 $\overline{7}$ 

1 Both operations are associative and commutative. Note that if MPI\_MAXLOC is applied  $\mathbf{2}$ to reduce a sequence of pairs  $(u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)$ , then the value returned is 3 (u, r), where  $u = \max_i u_i$  and r is the index of the first global maximum in the sequence. 4 Thus, if each process supplies a value and its rank within the group, then a reduce operation  $\mathbf{5}$ with  $op = MPI_MAXLOC$  will return the maximum value and the rank of the first process with 6 that value. Similarly, MPI\_MINLOC can be used to return a minimum and its index. More  $\overline{7}$ generally, MPI\_MINLOC computes a *lexicographic minimum*, where elements are ordered 8 according to the first component of each pair, and ties are resolved according to the second 9 component. 10 The reduce operation is defined to operate on arguments that consist of a pair: value  $^{11}$ and index. For both Fortran and C, types are provided to describe the pair. The potentially 12mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, 13for Fortran, by having the MPI-provided type consist of a pair of the same type as value, 14and coercing the index to this type also. In C, the MPI-provided pair type has distinct 15types and the index is an int. 16In order to use MPI\_MINLOC and MPI\_MAXLOC in a reduce operation, one must provide 17a datatype argument that represents a pair (value and index). MPI provides nine such 18predefined datatypes. The operations MPI\_MAXLOC and MPI\_MINLOC can be used with 19each of the following datatypes. 2021Fortran: Description Name 22pair of REALs MPI\_2REAL 23MPI\_2DOUBLE\_PRECISION pair of DOUBLE PRECISION variables  $^{24}$ MPI\_2INTEGER pair of INTEGERs 252627C: 28Name Description 29float and int 30 MPI\_FLOAT\_INT double and int MPI\_DOUBLE\_INT 31MPI\_LONG\_INT long and int 32 MPI\_2INT pair of int 33 MPI\_SHORT\_INT short and int 34MPI\_LONG\_DOUBLE\_INT long double and int 35 36 The datatype MPI\_2REAL is as if defined by the following (see Section 5.1). 37 38MPI\_Type\_contiguous(2, MPI\_REAL, MPI\_2REAL); 39 40Similar statements apply for MPI\_2INTEGER, MPI\_2DOUBLE\_PRECISION, and MPI\_2INT. 41 The datatype MPI\_SHORT\_INT is as if defined by the following sequence of instructions. 42struct mystruct { 43 short val; 44int rank; 45}; 46 type[0] = MPI\_SHORT; 47type[1] = MPI\_INT; 48

```
disp[0] = 0;
disp[1] = offsetof(struct mystruct, rank);
block[0] = 1;
block[1] = 1;
MPI_Type_create_struct(2, block, disp, type, MPI_SHORT_INT);
```

Similar statements apply for MPI\_FLOAT\_INT, MPI\_LONG\_INT and MPI\_DOUBLE\_INT. The following examples use intra-communicators.

**Example 6.17** Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```
. . .
/* each process has an array of 30 double: ain[30]
 */
double ain[30], aout[30];
int ind[30];
struct {
    double val;
    int
          rank;
} in[30], out[30];
int i, myrank, root;
MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}
MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
/* At this point, the answer resides on process root
 */
if (myrank == root) {
    /* read ranks out
     */
    for (i=0; i<30; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
    }
```

Example 6.18 Same example, in Fortran.
...
! each process has an array of 30 double: ain(30)
DOUBLE PRECISION ain(30), aout(30)
INTEGER ind(30)
DOUBLE PRECISION in(2,30), out(2,30)

}

1

2

4

5 6

7

8 9

10

11 12

13

14

15

16

17

18

19

20

21

22 23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39 40

 $41 \\ 42$ 

43

44 45

46

47

```
1
     INTEGER i, myrank, root, ierr
\mathbf{2}
3
     CALL MPI_COMM_RANK(comm, myrank, ierr)
4
     DO i=1,30
\mathbf{5}
         in(1,i) = ain(i)
6
        in(2,i) = myrank
                            ! myrank is coerced to a double
7
     END DO
8
9
     CALL MPI_REDUCE(in, out, 30, MPI_2DOUBLE_PRECISION, MPI_MAXLOC, root,&
10
                       comm, ierr)
11
     ! At this point, the answer resides on process root
12
13
     IF (myrank .EQ. root) THEN
14
         ! read ranks out
15
        DO i=1,30
16
            aout(i) = out(1,i)
17
            ind(i) = out(2,i) ! rank is coerced back to an integer
^{18}
        END DO
19
     END IF
20
21
     Example 6.19 Each process has a non-empty array of values. Find the minimum global
22
     value, the rank of the process that holds it and its index on this process.
23
^{24}
     #define LEN
                      1000
25
26
     float val[LEN];
                              /* local array of values */
27
                              /* local number of values */
     int count;
28
     int myrank, minrank, minindex;
29
     float minval;
30
31
     struct {
32
          float value;
33
          int
                index;
34
     } in, out;
35
36
          /* local minloc */
37
     in.value = val[0];
38
     in.index = 0;
39
     for (i=1; i < count; i++)
40
          if (in.value > val[i]) {
41
              in.value = val[i];
42
              in.index = i;
43
          }
44
45
          /* global minloc */
46
     MPI_Comm_rank(comm, &myrank);
47
     in.index = myrank*LEN + in.index;
```

```
MPI_Reduce(&in, &out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm);
    /* At this point, the answer resides on process root
    */
if (myrank == root) {
    /* read answer out
    */
    minval = out.value;
    minrank = out.index / LEN;
    minindex = out.index % LEN;
}
```

Rationale. The definition of MPI\_MINLOC and MPI\_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. By assigning a value other than myrank to the in.index field, a programmer can provide a different definition of MPI\_MAXLOC and MPI\_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (*End of rationale.*)

6.9.5 User-Defined Reduction Operations

```
MPI_OP_CREATE(user_fn, commute, op)
```

IN	user_fn	user defined function (function)
IN	commute	${\sf true} \ {\rm if} \ {\rm commutative}; \ {\sf false} \ {\rm otherwise}.$
OUT	ор	operation (handle)

#### C binding

EXTERNAL USER\_FN

```
int MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)
int MPI_Op_create_c(MPI_User_function_c *user_fn, int commute, MPI_Op *op)
Fortran 2008 binding
MPI_Op_create(user_fn, commute, op, ierror)
    PROCEDURE(MPI_User_function) :: user_fn
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Op_create_c(user_fn, commute, op, ierror) !(_c)
    PROCEDURE(MPI_User_function_c) :: user_fn
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_OP_CREATE(USER_FN, COMMUTE, OP, IERROR)
```

 $\mathbf{2}$ 

12	LOGICAL COMMUTE INTEGER OP, IERROR
3	
4	MPI_OP_CREATE binds a user-defined reduction operation to an
5	op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE,
6	MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN,
7	$MPI_EXSCAN$ , all nonblocking variants of those (see Section 6.12), and
8	MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute
9	= true, then the operation should be both commutative and associative. If commute $=$ false,
10	then the order of operands is fixed and is defined to be in ascending, process rank order,
11	beginning with process zero. The order of evaluation can be changed, talking advantage of
12	the associativity of the operation. If $commute = true$ then the order of evaluation can be
13	changed, taking advantage of commutativity and associativity.
14	In Fortran when using USE mpi_f08, the large count variant shall be called explicitly
14	as MPI_Op_create_c (i.e., with suffix "_c") because interface polymorphism cannot be used
16	to differentiate between the two different user callback prototypes despite their different
17	type signatures.
18	The argument user_fn is the user-defined function, which must have the following four
19	arguments: invec, inoutvec, len, and datatype.
20	MPI_USER_FUNCTION also supports large count types in separate additional MPI
21	callback function prototype declarations in C (suffixed with the "_c") and in Fortran when
22	using USE mpi_f08.
23	The ISO C prototypes for the functions are the following.
24	typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
25	<pre>MPI_Datatype *datatype);</pre>
26	<pre>typedef void MPI_User_function_c(void *invec, void *inoutvec,</pre>
27	MPI_Count *len, MPI_Datatype *datatype);
28	MI_count #ien, MI_Datatype #datatype),
29	The Fortran declarations of the user-defined function user_fn appear below.
30	ABSTRACT INTERFACE
31	SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
32	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
33	TYPE(C_PTR), VALUE :: invec, inoutvec
34	INTEGER :: len
35	TYPE(MPI_Datatype) :: datatype
36	ABSTRACT INTERFACE
37	SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype) !(_c)
38	USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
39	TYPE(C_PTR), VALUE :: invec, inoutvec
40	INTEGER(KIND=MPI_COUNT_KIND) :: len
41	TYPE(MPI_Datatype) :: datatype
42	
43	SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
44	<type> INVEC(LEN), INOUTVEC(LEN)</type>
45	INTEGER LEN, DATATYPE
46	The datatype argument is a handle to the datatype that was passed into the call to
47	MPI_REDUCE. The user reduce function should be written such that the following holds:
48	

Let  $u[0], \ldots, u[len-1]$  be the len elements in the communication buffer described by the arguments invec, len and datatype when the function is invoked; let  $v[0], \ldots, v[len-1]$  be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function is invoked; let  $w[0], \ldots, w[len-1]$  be len elements in the communication buffer described by the arguments inoutvec, len and datatype when the function returns; then  $w[i] = u[i] \circ v[i]$ , for  $i=0, \ldots$ , len-1, where  $\circ$  is the reduce operation that the function computes.

Informally, we can think of invec and inoutvec as arrays of len elements that user\_fn is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: i.e., the function returns in inoutvec[i] the value invec[i]  $\circ$  inoutvec[i], for i=0, ..., count-1, where  $\circ$  is the combining operation computed by the function.

*Rationale.* The len argument allows MPI\_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different datatypes. (*End of rationale.*)

When calling any reduction or prefix scan MPI procedure with a user-defined MPI operator, the type of the count parameter in the call to the reduction or prefix scan MPI procedure does not need to be identical to the type of the len parameter in the user function associated with the user-defined MPI operator. If the count parameter has a type of int in C or INTEGER in Fortran and the len parameter has a type of MPI\_COUNT, then MPI will perform the appropriate widening type conversion of the len parameter. If the count parameter has a type of MPI\_COUNT and the len parameter has a type of int in C or INTEGER in Fortran, then MPI will perform the appropriate narrowing type conversion of the len parameter. If this narrowing conversion would result in truncation of the len value, then MPI will call the user function multiple times with a sequence of values for len that sum to the value of count.

Advice to implementors. If the number of data items cannot be represented in len, the implementation may need to invoke user\_fn multiple times. (End of advice to implementors.)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. MPI\_ABORT may be called inside the function in case of an error.

Advice to users.Suppose one defines a library of user-defined reduce functions that43are overloaded: the datatype argument is used to select the right execution path at each44invocation, according to the types of the operands. The user-defined reduce function45cannot "decode" the datatype argument that it is passed, and cannot identify, by itself,46the correspondence between the datatype handles and the datatype they represent.47This correspondence was established when the datatypes were created.Before the48

 $\mathbf{5}$ 

 $^{24}$ 

	230	CHAPTER 6. COLLECTIVE COMMUNICATION
1		library is used, a library initialization preamble must be executed. This preamble
2		code will define the datatypes that are used by the library, and store handles to these
3		datatypes in global, static variables that are shared by the user code and the library
4		code.
5		The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using
6		the Fortran calling conventions and will pass a Fortran-type datatype argument; the
7		C version will use C calling convention and the C representation of a datatype handle.
8		Users who plan to mix languages should define their reduction functions accordingly.
9		(End of advice to users.)
10		(Linu of unoice to users.)
11		Advice to implementors. We outline below a naive and inefficient implementa-
12		tion of MPI_REDUCE not supporting the "in place" option and only valid for intra-
13		communicators.
14		communicators.
15		
16		<pre>MPI_Comm_size(comm, &amp;groupsize);</pre>
17		MPI_Comm_rank(comm, &rank);
18		if (rank > 0) {
19		<pre>MPI_Recv(tempbuf, count, datatype, rank-1,);</pre>
20		User_reduce(tempbuf, sendbuf, count, datatype);
21		}
22		<pre>if (rank &lt; groupsize-1) {</pre>
23		<pre>MPI_Send(sendbuf, count, datatype, rank+1,);</pre>
24		}
25		/* answer now resides in process groupsize-1 now send to root
26		*/
27		if (rank == root) {
28		<pre>MPI_Irecv(recvbuf, count, datatype, groupsize-1,, &amp;req);</pre>
29		}
30		<pre>if (rank == groupsize-1) {</pre>
31		<pre>MPI_Send(sendbuf, count, datatype, root,);</pre>
32		}
33		if (rank == root) {
34		<pre>MPI_Wait(&amp;req, &amp;status);</pre>
35		}
36		
37		The reduction computation proceeds, sequentially, from process 0 to process
38		groupsize-1. This order is chosen so as to respect the order of a possibly noncom-
39		mutative operator defined by the function User_reduce(). A more efficient imple-
40		mentation is achieved by taking advantage of associativity and using a logarithmic
41		tree reduction. Commutativity can be used to advantage, for those cases in which
42		the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary
43		buffer required can be reduced, and communication can be pipelined with computa-
44		tion, by transferring and reducing the elements in chunks of size len <count.< td=""></count.<>
45		
46		The predefined reduce operations can be implemented as a library of user-defined operations. However, better performance might be achieved if MPL REDUCE handles
47		operations. However, better performance might be achieved if MPI_REDUCE handles these functions as a special case. ( <i>End of advice to implementare</i> .)
48		these functions as a special case. (End of advice to implementors.)
10		

CHAPTER 6. COLLECTIVE COMMUNICATION

MPI_OP_FREE(op)	1
INOUT op operation (handle)	2
	3
C binding	4
int MPI_Op_free(MPI_Op *op)	6
Fortran 2008 binding	7
MPI_Op_free(op, ierror)	8
TYPE(MPI_Op), INTENT(INOUT) :: op	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
Fortran binding	12
MPI_OP_FREE(OP, IERROR) INTEGER OP, IERROR	13
INTEGER OF, TEMIOR	14
Marks a user-defined reduction operation for deallocation and sets $op$ to MPI_OP_NU	JLL. 15 16
Example of User-Defined Reduce	17
	18
It is time for an example of user-defined reduction. The example in this section uses intra-communicator.	an 19 20
	20
<b>Example 6.20</b> Compute the product of an array of complex numbers, in C.	22
typedef struct {	23
double real, imag;	24
} Complex;	25
	26
/* the user-defined function	27
<pre>*/ void myProd(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)</pre>	28
{	29
int i;	30
Complex c;	31 32
Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;	33
	34
for (i=0; i< *len; ++i) {	35
c.real = inout->real*in->real -	36
<pre>inout-&gt;imag*in-&gt;imag;</pre>	37
c.imag = inout->real*in->imag +	38
<pre>inout-&gt;imag*in-&gt;real;</pre>	39
<pre>*inout = c; intt: inout++:</pre>	40
<pre>in++; inout++; }</pre>	41
}	42
	43
/* and, to call it	44
*/	45
	40
	48

```
/* each process has an array of 100 Complexes
 */
Complex a[100], answer[100];
MPI_Op myOp;
MPI_Datatype ctype;
/* explain to MPI how type Complex is defined
*/
MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
MPI_Type_commit(&ctype);
/* create the complex-product user-op
*/
MPI_Op_create(myProd, 1, &myOp);
MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);
/* At this point, the answer, which consists of 100 Complexes,
 * resides on process root
 */
```

**Example 6.21** How to use the mpi\_f08 interface of the Fortran MPI\_User\_function.

```
subroutine my_user_function(invec, inoutvec, len, type) bind(c)
use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
use mpi_f08
type(c_ptr), value :: invec, inoutvec
integer :: len
type(MPI_Datatype) :: type
real, pointer :: invec_r(:), inoutvec_r(:)
if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
call c_f_pointer(invec, invec_r, (/ len /))
call c_f_pointer(inoutvec, inoutvec_r, (/ len /))
inoutvec_r = invec_r + inoutvec_r
end if
end subroutine
```

#### 6.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes
 in a group. MPI requires that all processes from the same group participating in these
 operations receive identical results.

 $^{31}$ 

MPI	_ALLREDU	JCE(sendbuf, recvbuf, count,	datatype, op, comm)	1
IN	sen	ndbuf	starting address of send buffer (choice)	2 3
OI	JT rec	vbuf	starting address of receive buffer (choice)	4
IN	cou		number of elements in send buffer (non-negative integer)	5 6
IN	dat		datatype of elements of send buffer (handle)	7
IN			operation (handle)	8
	•			9 10
IN	con	nm	communicator (handle)	11
Сh	inding			12
		educe(const void *sendb	uf, void *recvbuf, int count,	13
1110			, MPI_Op op, MPI_Comm comm)	14
÷				15
int	MP1_ALLIE		dbuf, void *recvbuf, MPI_Count count, , MPI_Op op, MPI_Comm comm)	16 17
				17
	tran 2008	0		19
MP1_			nt, datatype, op, comm, ierror)	20
		DIMENSION(), INTENT( DIMENSION() :: recvb		21
		INTENT(IN) :: count	ui -	22
		_Datatype), INTENT(IN)	:: datatype	23
	TYPE(MPI_	_Op), INTENT(IN) :: op		24
	TYPE(MPI_	_Comm), INTENT(IN) :: c	omm	25 26
	INTEGER,	OPTIONAL, INTENT(OUT)	:: ierror	20
MPI_	Allreduce	e(sendbuf, recvbuf, cou	nt, datatype, op, comm, ierror) !(_c)	28
		DIMENSION(), INTENT(		29
		DIMENSION() :: recvb		30
		(IND=MPI_COUNT_KIND), I		31
		_Datatype), INTENT(IN)	:: datatype	32
		_Op), INTENT(IN) :: op _Comm), INTENT(IN) :: c		33
		_COMMINITY, INTENT(IN) C OPTIONAL, INTENT(OUT)		$\frac{34}{35}$
_	-	-		36
	tran bindi			37
MP1_			NT, DATATYPE, OP, COMM, IERROR)	38
		ENDBUF(*), RECVBUF(*) COUNT, DATATYPE, OP, CO	MM TERROR	39
				40
		,	API_ALLREDUCE behaves the same as	41
MPI	_REDUCE (	except that the result appea	ars in the receive buffer of all the group members.	42 43
	Advice to	<i>implementors.</i> The all-	reduce operations can be implemented as a re-	43 44
		-	ever, a direct implementation can lead to better	45
		nce. (End of advice to impl		46
		-		47
				48

#### MPL ALLREDUCE(sendbuf recybuf count datatype op comm)

The "in place" option for intra-communicators is specified by passing the value MPI\_IN\_PLACE to the argument sendbuf at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data. If comm is an inter-communicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

The following example uses an intra-communicator.

240

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11

18

21

22

23

 $^{24}$ 

25

27

2829

30

 $^{31}$ 

32 33

3435

36

37

38

**Example 6.22** A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 6.16).

```
12
     SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
13
     REAL a(m), b(m,n)
                           ! local slice of array
14
     REAL c(n)
                            ! result
15
     REAL sum(n)
16
     INTEGER n, comm, i, j, ierr
17
     ! local sum
19
     DO j=1,n
20
        sum(j) = 0.0
        DO i=1,m
           sum(j) = sum(j) + a(i)*b(i,j)
        END DO
     END DO
26
     ! global sum
     CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)
     ! return result at all nodes
     RETURN
     END
```

#### Process-Local Reduction 6.9.7

The functions in this section are of importance to library implementors who may want to implement special reduction patterns that are otherwise not easily covered by the standard MPI operations.

The following function applies a reduction operator to local arguments.

MPI_RED	UCE_LOCAL(inbuf, inoutbuf, c	count, datatype, op)	1
IN	inbuf	input buffer (choice)	2
INOUT	inoutbuf	combined input and output buffer (choice)	3 4
IN	count	number of elements in inbuf and inoutbuf buffers	5
	count	(non-negative integer)	6
IN	datatype	datatype of elements of inbuf and inoutbuf buffers	7
	adatype	(handle)	8
IN	ор	operation (handle)	9 10
	<b>с</b> р	operation (namelo)	10
C bindin	g		12
	-	inbuf, void *inoutbuf, int count,	13
	MPI_Datatype datatyp	pe, MPI_Op op)	14
int MPI H	Reduce local c(const void	<pre>*inbuf, void *inoutbuf, MPI_Count count,</pre>	15
	MPI_Datatype datatyp		16
Fortron (	2008 binding		17 18
	6	count, datatype, op, ierror)	19
	(*), DIMENSION(), INTEN		20
	(*), DIMENSION() :: ino		21
	GER, INTENT(IN) :: count		22
	(MPI_Datatype), INTENT(IN		23
	(MPI_Op), INTENT(IN) :: o	-	24
INTEC	GER, OPTIONAL, INTENT(OUT	) :: ierror	25 26
MPI_Reduo	ce_local(inbuf, inoutbuf,	<pre>count, datatype, op, ierror) !(_c)</pre>	27
	(*), DIMENSION(), INTEN		28
	(*), DIMENSION() :: ino		29
	GER(KIND=MPI_COUNT_KIND),		30
	(MPI_Datatype), INTENT(IN (MPI_Op), INTENT(IN) :: o		31
	GER, OPTIONAL, INTENT(OUT	-	32
		,	33 34
Fortran I	_		35
	<pre>&gt; INBUF(*), INOUTBUF(*)</pre>	COUNT, DATATYPE, OP, IERROR)	36
	GER COUNT, DATATYPE, OP,	IERROR	37
			38
ine f	unction applies the operation	given by op element-wise to the elements of inbuf	20

The function applies the operation given by **op** element-wise to the elements of inbuf and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined operations in Section 6.9.5. Both inbuf and inoutbuf (input as well as result) have the same number of elements given by count and the same datatype given by datatype. The MPI\_IN\_PLACE option is not allowed.

Reduction operations can be queried for their commutativity.

39

40

 $^{41}$ 

42

43

```
1
     MPI_OP_COMMUTATIVE(op, commute)
\mathbf{2}
       IN
                                              operation (handle)
                 ор
3
       OUT
                 commute
                                              true if op is commutative, false otherwise (logical)
4
5
6
     C binding
7
     int MPI_Op_commutative(MPI_Op op, int *commute)
8
     Fortran 2008 binding
9
     MPI_Op_commutative(op, commute, ierror)
10
          TYPE(MPI_Op), INTENT(IN) :: op
11
          LOGICAL, INTENT(OUT) :: commute
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
16
          INTEGER OP, IERROR
17
          LOGICAL COMMUTE
18
19
             Reduce-Scatter
     6.10
20
21
     MPI includes variants of the reduce operations where the result is scattered to all processes
22
     in a group on return. One variant scatters equal-sized blocks to all processes, while another
23
     variant scatters blocks that may vary in size for each process.
^{24}
25
26
     6.10.1 MPI_REDUCE_SCATTER_BLOCK
27
28
29
     MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm)
30
       IN
                 sendbuf
                                              starting address of send buffer (choice)
^{31}
       OUT
                 recvbuf
                                              starting address of receive buffer (choice)
32
33
       IN
                                              element count per block (non-negative integer)
                 recvcount
34
                                              datatype of elements of send and receive buffers
       IN
                 datatype
35
                                              (handle)
36
       IN
37
                 ор
                                              operation (handle)
38
       IN
                                              communicator (handle)
                 comm
39
40
     C binding
41
     int MPI_Reduce_scatter_block(const void *sendbuf, void *recvbuf,
42
                     int recvcount, MPI_Datatype datatype, MPI_Op op,
43
                     MPI_Comm comm)
44
     int MPI_Reduce_scatter_block_c(const void *sendbuf, void *recvbuf,
45
46
                     MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
47
                     MPI_Comm comm)
48
```

Fortran 2008 binding	1
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,	2
ierror)	3
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	4
TYPE(*), DIMENSION() :: recvbuf	5
INTEGER, INTENT(IN) :: recvcount	6
TYPE(MPI_Datatype), INTENT(IN) :: datatype	7
TYPE(MPI_Op), INTENT(IN) :: op	8
TYPE(MPI_Comm), INTENT(IN) :: comm	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,	11
ierror) !(_c)	12
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	13
TYPE(*), DIMENSION() :: recvbuf	14
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount	15
TYPE(MPI_Datatype), INTENT(IN) :: datatype	16
TYPE(MPI_Op), INTENT(IN) :: op	17
TYPE(MPI_Comm), INTENT(IN) :: comm	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
	20
Fortran binding	21
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,	22
IERROR)	23
<type> SENDBUF(*), RECVBUF(*)</type>	24
INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR	25
	26

If comm is an intra-communicator, MPI\_REDUCE\_SCATTER\_BLOCK first performs a global, element-wise reduction on vectors of  $count = n^* recvcount$  elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcount, datatype, op and comm. The resulting vector is treated as n consecutive blocks of recvcount elements that are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcount, and datatype.

The MPI\_REDUCE\_SCATTER\_BLOCK routine is func-Advice to implementors. tionally equivalent to: an MPI\_REDUCE collective operation with count equal to recvcount\*n, followed by an MPI\_SCATTER with sendcount equal to recvcount. However, a direct implementation may run faster. (End of advice to implementors.)

The "in place" option for intra-communicators is specified by passing MPI\_IN\_PLACE in the sendbuf argument on all processes. In this case, the input data is taken from the receive buffer.

If comm is an inter-communicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B) and vice versa. Within each group, all processes provide the same value for the recvcount argument, and provide input vectors of  $count = n^{*}recvcount$  elements stored in the send buffers, where n is the size of the group. The number of elements count must be the same

27

28

29

30

31

32

33

34 35

36

37

38

39 40

41

42

43

44

45

46

47

1 2 3		groups. The resulting vector lements among the processes if	or from the other group is scattered in blocks of in the group.
4 5 6 7 8	Other	roup can be determined by t	s needed so that the length of the send buffer of the local recvcount argument of the other group. ded to figure out how many elements are reduced.
9	6.10.2 M	PI_REDUCE_SCATTER	
10 11 12 13 14	such that the		unctionality of MPI_REDUCE_SCATTER_BLOCK i size. Block sizes are determined by the recvcounts ecvcounts[i] elements.
15	MPI_REDU	CE_SCATTER(sendbuf, recvbi	ıf, recvcounts, datatype, op, comm)
16 17	IN	sendbuf	starting address of send buffer (choice)
18	OUT	recvbuf	starting address of receive buffer (choice)
19 20 21	IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements of the result distributed to each process.
22 23 24	IN	datatype	datatype of elements of send and receive buffers (handle)
25	IN	ор	operation (handle)
26 27	IN	comm	communicator (handle)
28 29 30 31 32 33		educe_scatter(const void const int recvcounts  MPI_Comm comm) educe_scatter_c(const voi	[], MPI_Datatype datatype, MPI_Op op, d *sendbuf, void *recvbuf,
34 35		const MPI_Count recvo MPI_Op op, MPI_Comm o	<pre>counts[], MPI_Datatype datatype, comm)</pre>
36 37 38 39 40	MPI_Reduce	008 binding e_scatter(sendbuf, recvbu ierror) *), DIMENSION(), INTENT	f, recvcounts, datatype, op, comm, (IN) :: sendbuf
41 42	INTEG	*), DIMENSION() :: recv ER, INTENT(IN) :: recvcou MPI_Datatype), INTENT(IN)	nts(*)
43 44 45 46	TYPE(I TYPE(I	MPI_Op), INTENT(IN) :: op MPI_Comm), INTENT(IN) :: ER, OPTIONAL, INTENT(OUT)	comm
47 48	MPI_Reduce	e_scatter(sendbuf, recvbu ierror) !(_c)	f, recvcounts, datatype, op, comm,

```
TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                         1
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                         \mathbf{2}
                                                                                         3
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                         4
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                         5
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                         6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
                                                                                         9
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
                                                                                         10
               IERROR)
                                                                                         11
    <type> SENDBUF(*), RECVBUF(*)
                                                                                         12
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
                                                                                         13
    If comm is an intra-communicator, MPI_REDUCE_SCATTER first performs a global,
                                                                                         14
element-wise reduction on vectors of count = \sum_{i=0}^{n-1} recvcounts[i] elements in the send buffers
                                                                                         15
                                                                                         16
defined by sendbuf, count and datatype, using the operation op, where n is the number of
                                                                                         17
processes in the group of comm. The routine is called by all group members using the
```

processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI\_REDUCE\_SCATTER routine is functionally equivalent to: an MPI\_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI\_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intra-communicators is specified by passing MPI\_IN\_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the "in place" option on all processes, since the processes for which recvcounts[i] ==0 may not have allocated a receive buffer.

If comm is an inter-communicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B), and vice versa. Within each group, all processes provide the same recvcounts argument, and provide input vectors of count =  $\sum_{i=0}^{n-1} \text{recvcounts}[i]$  elements stored in the send buffers, where n is the size of the group. The resulting vector from the other group is scattered in blocks of recvcounts[i] elements among the processes in the group. The number of elements count must be the same for the two groups.

*Rationale.* The last restriction is needed so that the length of the send buffer can be determined by the sum of the local **recvcounts** entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

### 6.11 Scan

245

18

19

20

21

22

23

 $^{24}$ 

25

26 27

28

29

30

31

32

33

34

35

36

37

38 39

40

41

42 43 44

```
246
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.11.1 Inclusive Scan
\mathbf{2}
3
4
     MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)
5
       IN
                sendbuf
                                            starting address of send buffer (choice)
6
       OUT
                recvbuf
                                            starting address of receive buffer (choice)
7
8
       IN
                                            number of elements in input buffer (non-negative
                count
9
                                            integer)
10
       IN
                                            datatype of elements of input buffer (handle)
                datatype
11
       IN
                                            operation (handle)
                op
12
13
       IN
                comm
                                            communicator (handle)
14
15
     C binding
16
     int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
17
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
18
     int MPI_Scan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
19
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
20
21
     Fortran 2008 binding
22
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
23
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
24
         TYPE(*), DIMENSION(...) :: recvbuf
25
         INTEGER, INTENT(IN) :: count
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Op), INTENT(IN) :: op
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
^{31}
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
32
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
33
         TYPE(*), DIMENSION(...) :: recvbuf
34
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Op), INTENT(IN) :: op
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
```

```
40 Fortran binding
```

<sup>44</sup> If comm is an intra-communicator, MPI\_SCAN is used to perform a prefix reduction <sup>45</sup> on data distributed across the group. The operation returns, in the receive buffer of the <sup>46</sup> process with rank i, the reduction of the values in the send buffers of processes with ranks <sup>47</sup> 0,...,i (inclusive). The routine is called by all group members using the same arguments <sup>48</sup> for count, datatype, op and comm, except that for user-defined operations, the same rules

		The type of operations supported, their semantics, and the	$\frac{1}{2}$
		re buffers are as for MPI_REDUCE. r intra-communicators is specified by passing MPI_IN_PLACE	3
		his case, the input data is taken from the receive buffer, and	4
	by the output data.	his case, the input data is taken from the receive build, and	5
-	• •	for inter-communicators.	6
1 11			7
6 11 2	Exclusive Scan		8
0.11.2			9
			10
MPI_EX	SCAN(sendbuf, recvbu	f, count, datatype, op, comm)	11
IN	sendbuf	starting address of send buffer (choice)	12 13
OUT	recvbuf	starting address of receive buffer (choice)	14
			15
IN	count	number of elements in input buffer (non-negative integer)	16 17
IN	datatype	datatype of elements of input buffer (handle)	18
IN		operation (handle)	19
	ор		20
IN	comm	intra-communicator (handle)	21
			22
C bind	0		23
int MP		*sendbuf, void *recvbuf, int count,	24
	MP1_Datatype	e datatype, MPI_Op op, MPI_Comm comm)	25
int MP	I_Exscan_c(const vo	id *sendbuf, void *recvbuf, MPI_Count count,	26
	MPI_Datatype	e datatype, MPI_Op op, MPI_Comm comm)	27
Fontro	- 2008 binding		28
	n 2008 binding	uf, count, datatype, op, comm, ierror)	29
		), INTENT(IN) :: sendbuf	30
	PE(*), DIMENSION(		31
	<pre>FEGER, INTENT(IN) :</pre>		32
		NTENT(IN) :: datatype	33 34
	PE(MPI_Op), INTENT(		35
	PE(MPI_Comm), INTEN	•	36
		TENT(OUT) :: ierror	37
NDT D	( )) ( )		38
		uf, count, datatype, op, comm, ierror) !(_c)	39
	PE(*), DIMENSION( PE(*), DIMENSION(	), INTENT(IN) :: sendbuf	40
		T_KIND), INTENT(IN) :: count	41
		NTENT(IN) :: datatype	42
	PE(MPI_Op), INTENT(		43
	PE(MPI_Comm), INTEN	-	44
		TENT(OUT) :: ierror	45
			46
	n binding		47
MPI_EXS	SCAN(SENDBUF, RECVB	UF, COUNT, DATATYPE, OP, COMM, IERROR)	48

<pre>1 <type> SENDBUF(*), RECVBUF(*) 2 INTEGER COUNT, DATATYPE, OP, COMM, IERROR</type></pre>
3
<sup>4</sup> If comm is an intra-communicator, MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is
undefined and recycluf is not significant on process 0. The value in recycluf on the process
$_{6}^{6}$ with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes
with rank $i > 1$ , the operation returns, in the receive buffer of the process with rank $i$ , the
$_{9}$ reduction of the values in the send buffers of processes with ranks $0, \ldots, i-1$ (inclusive). The
routine is called by all group members using the same arguments for count, datatype, op and
<sup>11</sup> comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE.
The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.
The "in place" option for intra-communicators is specified by passing MPLIN PLACE
in the sendbuf argument. In this case, the input data is taken from the receive buffer, and
$_{15}$ replaced by the output data. The receive buffer on rank 0 is not changed by this operation.
This operation is invalid for inter-communicators.
<sup>19</sup> <i>Rationale.</i> The exclusive scan is more general than the inclusive scan. Any inclusive
scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the
exclusive scan cannot be computed with the inclusive scan. ( <i>End of rationale.</i> )
22
<sup>23</sup> <sub>24</sub> 6.11.3 Example using MPI_SCAN
<sup>25</sup> The example in this section uses an intra-communicator.
<sup>26</sup>
Example 6.23 This example uses a user-defined operation to produce a <i>segmented scan</i> . A
segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate
<sup>29</sup> the various segments of the scan. For example:
$values v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8$
$\begin{array}{c ccccccccccccccccccccccccccccccccccc$
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
24
The operator that produces this effect is The operator that produces the operator the operator that produces the operator the operator the operator that produces the operator the
$\begin{pmatrix} u \\ \end{pmatrix} \begin{pmatrix} v \\ \end{pmatrix} \begin{pmatrix} w \\ \end{pmatrix}$
$\begin{pmatrix} 36\\ 37 \end{pmatrix} \circ \begin{pmatrix} v\\ j \end{pmatrix} = \begin{pmatrix} w\\ j \end{pmatrix},$
38 ( / ( / ) ( / ) / ) (
40
40
40

. . .

```
typedef struct {
    double val;
    int log;
} SegScanPair;
/* the user-defined function
 */
void segScan(SegScanPair *in, SegScanPair *inout, int *len,
             MPI_Datatype *dptr)
{
    int i;
    SegScanPair c;
    for (i=0; i< *len; ++i) {</pre>
        if (in->log == inout->log)
             c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}
Note that the inout argument to the user-defined function corresponds to the right-hand
operand of the operator. When using this operator, we must be careful to specify that it is
noncommutative, as in the following.
    int i,base;
    SegScanPair a, answer;
    MPI_Op
                  myOp;
    MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
    MPI_Aint
                  disp[2];
                  blocklen[2] = { 1, 1};
    int
    MPI_Datatype sspair;
    /* explain to MPI how type SegScanPair is defined
     */
    MPI_Get_address(&a, disp);
    MPI_Get_address(&a.log, disp+1);
    base = disp[0];
    for (i=0; i<2; ++i) disp[i] -= base;</pre>
    MPI_Type_create_struct(2, blocklen, disp, type, &sspair);
    MPI_Type_commit(&sspair);
    /* create the segmented-scan user-op
     */
    MPI_Op_create(segScan, 0, &myOp);
```

1

 $\mathbf{2}$ 

3

4

5

7

8

9

10

11

12

13 14

15

16

17

18

19

20

21

22

23

24 25

26

27 28

29

30

 $^{31}$ 

32

33

34

35 36

37

38

39

40

41

42

43

44

45

46

47

2 3

45

7

8

9

MPI\_Scan(&a, &answer, 1, sspair, myOp, comm);

### 6.12 Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by over-6 lapping communication and computation, and many systems enable this. Nonblocking collective operations combine the potential benefits of nonblocking point-to-point operations, to exploit overlap and to avoid synchronization, with the optimized implementation and message scheduling provided by collective operations [34, 38]. One way of doing this 10 would be to perform a blocking collective operation in a separate thread. An alternative 11 mechanism that often leads to better performance (e.g., avoids context switching, scheduler 12overheads, and thread management) is to use nonblocking collective communication [36]. 13

The nonblocking collective communication model is similar to the model used for non-14blocking point-to-point communication. A nonblocking call initiates a collective operation, 15which must be completed in a separate completion call. Once initiated, the operation 16may progress independently of any computation or other communication at participating 17processes. In this manner, nonblocking collective operations can mitigate possible synchro-18 nizing effects of collective operations by running them in the "background." In addition to 19 enabling communication-computation overlap, nonblocking collective operations can per-20form collective operations on overlapping communicators, which would lead to deadlocks 21with blocking operations. Their semantic advantages can also be useful in combination with 22point-to-point communication. 23

As in the nonblocking point-to-point case, all calls are local and return immediately,  $^{24}$ irrespective of the status of other processes. The call initiates the operation, which indicates 25that the system may start to copy data out of the send buffer and into the receive buffer. 26Once initiated, all associated send buffers and buffers associated with input arguments (such 27as arrays of counts, displacements, or datatypes in the vector versions of the collectives) 28should not be modified, and all associated receive buffers should not be accessed, until the 29 collective operation completes. The call returns a request handle, which must be passed to 30 a completion call.  $^{31}$ 

All completion calls (e.g., MPI\_WAIT) described in Section 3.7.3 are supported for 32 nonblocking collective operations. Similarly to the blocking case, nonblocking collective 33 operations are considered to be complete when the local part of the operation is finished, 34i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be 35 safely accessed and modified. Completion does not indicate that other processes have 36 completed or even started the operation (unless otherwise implied by the description of 37 the operation). Completion of a particular nonblocking collective operation also does not 38 indicate completion of any other posted nonblocking collective (or send-receive) operations, 39 whether they are posted before or after the completed operation. 40

- 41
- 4243

Advice to users. Users should be aware that implementations are allowed, but not required (with exception of MPI\_IBARRIER), to synchronize processes during the completion of a nonblocking collective operation. (End of advice to users.)

4445

Upon returning from a completion call in which a nonblocking collective operation 46completes, the values of the MPI\_SOURCE and MPI\_TAG fields in the associated status object, 47if any, are undefined. The value of MPI\_ERROR may be defined, if appropriate, according 48to the specification in Section 3.2.5. It is valid to mix different request types (i.e., any combination of collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI\_WAITALL). It is erroneous to call MPI\_REQUEST\_FREE or MPI\_CANCEL for a request associated with a nonblocking collective operation. Nonblocking collective requests created using the APIs described in this section are not persistent. However, persistent collective requests can be created using persistent collective operations described in Sections 6.13 and 8.8.

Rationale. Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (*End of rationale.*)

Multiple nonblocking collective operations can be outstanding on a single communicator. If the nonblocking call causes some system resource to be exhausted, then it will fail and raise an error. Quality implementations of MPI should ensure that this happens only in pathological cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with the ordering rules for blocking collective operations in threaded environments.

*Rationale.* Matching blocking and nonblocking collective operations is not allowed because the implementation might use different communication algorithms for the two cases. Blocking collective operations may be optimized for minimal time to completion, while nonblocking collective operations may balance time to completion with CPU overhead and asynchronous progression.

The use of tags for collective operations can prevent certain hardware optimizations. (*End of rationale.*)

Advice to users. If program semantics require matching blocking and nonblocking collective operations, then a nonblocking collective operation can be initiated and immediately completed with a blocking wait to emulate blocking behavior. (*End of advice to users.*)

In terms of data movement, each nonblocking collective operation has the same effect as its blocking counterpart for intra-communicators and inter-communicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the "in place" option is allowed exactly as described for the corresponding blocking collective operations. When using the "in place" option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

*Progression* rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

1 2

3

4

 $\mathbf{5}$ 

6 7

8

9

10

11 12

13

14

15

16

17

18

19

20

21

22

23

24 25

26

27

28

29

30

 $^{31}$ 

32 33

34

35

36

37

38 39

40 41

42

43

44

 $45 \\ 46$ 

47

<ul> <li>6.12.1 Nonblocking Barrier Synchronization</li> <li>MPI_IBARRIER(comm, request)</li> <li>IN comm communicator (handle)</li> <li>OUT request communication request (handle)</li> <li>OUT request strequest (handle)</li> <li>Fortran 2008 binding</li> <li>MPI_Ibarrier(comm, request, ierror)</li> <li>MPI_Ibarrier(comm, request, ierror)</li> <li>TYPE(MPI_Comm), INTENT(IN) :: comm</li> </ul>
MPI_IBARRIER(comm, request) IN comm communicator (handle) OUT request communication request (handle) C binding int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request) Fortran 2008 binding MPI_Ibarrier(comm, request, ierror) MPI_Ibarrier(comm), INTENT(IN) :: comm
<sup>9</sup> IN comm communicator (handle) OUT request communication request (handle) C binding int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request) Fortran 2008 binding MPI_Ibarrier(comm, request, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm
11       OUT request       communication request (handle)         12       13       C binding         14       int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)         15       Fortran 2008 binding         17       MPI_Ibarrier(comm, request, ierror)         18       TYPE(MPI_Comm), INTENT(IN) :: comm
<sup>12</sup> <sup>13</sup> C binding <sup>14</sup> int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request) <sup>15</sup> <sup>16</sup> Fortran 2008 binding <sup>17</sup> MPI_Ibarrier(comm, request, ierror) <sup>18</sup> TYPE(MPI_Comm), INTENT(IN) :: comm
C binding int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request) Fortran 2008 binding MPI_Ibarrier(comm, request, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm
<ul> <li>Fortran 2008 binding</li> <li>MPI_Ibarrier(comm, request, ierror)</li> <li>TYPE(MPI_Comm), INTENT(IN) :: comm</li> </ul>
18 TYPE(MPI_Comm), INTENT(IN) :: comm
19 TYPE(MPI_Request), INTENT(OUT) :: request
20 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
<sup>21</sup> <sub>22</sub> Fortran binding
MPI_IBARRIER(COMM, REQUEST, IERROR)
INTEGER COMM, REQUEST, IERROR
<sup>25</sup> MPI_IBARRIER is a nonblocking version of MPI_BARRIER. By calling MPI_IBARRIE
$^{26}$ a process notifies that it has reached the barrier. The call returns immediately, independent of the second seco
<sup>27</sup> dent of whether other processes have called MPI_IBARRIER. The usual barrier semanti
<sup>28</sup> are enforced at the corresponding completion operation (test or wait), which in the intr
<ul> <li>communicator case will complete only after all other processes in the communicator has</li> <li>called MPI_IBARRIER. In the inter-communicator case, it will complete when all process</li> </ul>
<ul> <li><sup>30</sup> called MPI_IBARRIER. In the inter-communicator case, it will complete when all process</li> <li><sup>31</sup> in the remote group have called MPI_IBARRIER.</li> </ul>
32
Advice to users. A nonblocking barrier can be used to hide latency. Moving indepen-
dent computations between the MPI_IBARRIER and the subsequent completion ca
can overlap the barrier latency and therefore shorten possible waiting times. The s
$_{36}$ mantic properties are also useful when mixing collective operations and point-to-point $(E - l - l - l - l - l - l - l - l - l - $
messages. (End of advice to users.)
38 39
40
41
42
43
44
45
46
47 48

CHAPTER 6. COLLECTIVE COMMUNICATION

6.12.2 N	onblocking Broadcast		1
			2
	CT/huffer count dotations to	at commencest)	4
	ST(buffer, count, datatype, roo	. ,	5
INOUT	buffer	starting address of buffer (choice)	6
IN	count	number of entries in buffer (non-negative integer)	7
IN	datatype	datatype of buffer (handle)	8 9
IN	root	rank of broadcast root (integer)	9 10
IN	comm	communicator (handle)	11
OUT	request	communication request (handle)	12
			13 14
C binding			14
int MPI_1	bcast(void *buffer, int MPI_Comm comm, MPI_R	count, MPI_Datatype datatype, int root, equest *request)	16 17
int MPI_I	<pre>Ebcast_c(void *buffer, MP</pre>	I_Count count, MPI_Datatype datatype,	18
	int root, MPI_Comm c	comm, MPI_Request *request)	19
Fortran 2	2008 binding		20 21
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)			
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buffer			22 23
INTEGER, INTENT(IN) :: count, root			24
	(MPI_Datatype), INTENT(IN		25
	<pre>[MPI_Comm), INTENT(IN) :: [MPI_Request), INTENT(OUT</pre>		26
	ER, OPTIONAL, INTENT(OUT	-	27
			28
		e, root, comm, request, ierror) !(_c)	29 30
	(*), DIMENSION(), ASYNC ER(KIND=MPI_COUNT_KIND),		31
	(MPI_Datatype), INTENT(IN		32
	ER, INTENT(IN) :: root	, aucusype	33
	MPI_Comm), INTENT(IN) ::	comm	34
TYPE	MPI_Request), INTENT(OUT	) :: request	35
INTEC	ER, OPTIONAL, INTENT(OUT	) :: ierror	36
Fortran k	binding		37 38
		E, ROOT, COMM, REQUEST, IERROR)	39
<type< td=""><td>&gt; BUFFER(*)</td><td></td><td>40</td></type<>	> BUFFER(*)		40
INTEG	ER COUNT, DATATYPE, ROOT	, COMM, REQUEST, IERROR	41
This c	call starts a nonblocking varia	nt of $MPI_BCAST$ (see Section 6.4).	42
			43
Example us	sing MPI_IBCAST		44
The evam	ple in this section uses an intr	a-communicator	45
THE CRAIN		a communicator.	46 47
			-11

1 **Example 6.24** Start a broadcast of 100 ints from process 0 to every process in the  $\mathbf{2}$ group, perform some computation on independent data, and then complete the outstanding 3 broadcast operation. 4 5MPI\_Comm comm; 6 int array1[100], array2[100]; 7 int root=0; 8 MPI\_Request req; 9 . . . 10 MPI\_Ibcast(array1, 100, MPI\_INT, root, comm, &req); 11 compute(array2, 100); 12MPI\_Wait(&req, MPI\_STATUS\_IGNORE); 13 146.12.3 Nonblocking Gather 151617MPI\_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, 18 request) 1920IN sendbuf starting address of send buffer (choice) 21sendcount number of elements in send buffer (non-negative IN 22 integer) 23sendtype datatype of send buffer elements (handle) IN  $^{24}$ 25OUT recvbuf address of receive buffer (choice, significant only at 26root) 27IN recvcount number of elements for any single receive 28(non-negative integer, significant only at root) 29 IN recvtype datatype of recv buffer elements (handle, significant 30 only at root)  $^{31}$ 32 IN rank of receiving process (integer) root 33 IN communicator (handle) comm 34 OUT request communication request (handle) 35 36 37 C binding int MPI\_Igather(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 3839 void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm, MPI\_Request \*request) 40 41 int MPI\_Igather\_c(const void \*sendbuf, MPI\_Count sendcount, 42MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 43 MPI\_Datatype recvtype, int root, MPI\_Comm comm, 44 MPI\_Request \*request) 4546Fortran 2008 binding 47MPI\_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 48 root, comm, request, ierror)

TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount, root TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request, ierror) !(\_c) TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: root TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI\_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR This call starts a nonblocking variant of  $MPI_GATHER$  (see Section 6.5). 

MPI_IGAT	HERV(sendbuf, sendcour comm, request)	nt, sendtype, recvbuf, recvcounts, displs, recvtype, root,		
IN	sendbuf	starting address of send buffer (choice)		
IN	sendcount	number of elements in send buffer (non-negative integer)		
IN	sendtype	datatype of send buffer elements (handle)		
OUT	recvbuf	address of receive buffer (choice, significant only at root)		
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)		
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)		
IN	recvtype	datatype of recv buffer elements (handle, significant only at root)		
IN	root	rank of receiving process (integer)		
IN	comm	communicator (handle)		
OUT	request	communication request (handle)		
<sup>25</sup> <sup>26</sup> C binding <sup>27</sup> int MPI_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, <sup>28</sup> void *recvbuf, const int recvcounts[], const int displs[], <sup>29</sup> MPI_Datatype recvtype, int root, MPI_Comm comm, <sup>30</sup> MPI_Request *request) <sup>31</sup> int MPI_Igatherv_c(const void *sendbuf, MPI_Count sendcount, <sup>32</sup> MPI_Datatype sendtype, void *recvbuf, <sup>33</sup> const MPI_Count recvcounts[], const MPI_Aint displs[],				
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)				
<pre>Fortran 2008 binding MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,</pre>				
	IN IN OUT IN IN IN IN IN IN OUT C bindin int MPI_: int MPI_: IN Fortran : MPI_IgatI TYPE INTEG TYPE INTEG TYPE	comm, request)INsendbufINsendcountINsendtypeOUTrecvbufINrecvcountsINdisplsINrecvtypeINrootINcommOUTrequestC bindingint MPI_Igatherv(const void * void *recvbuf, MPI_Datatype re MPI_Request *reint MPI_Igatherv_c(const void * const MPI_Count MPI_Datatype se const MPI_Count MPI_Request *refortran 2008 bindingMPI_Igatherv(sendbuf, sendcourcectype, root, TYPE(*), DIMENSION(), INTEGER, INTENT(IN) :: se TYPE(*), DIMENSION(), INTEGER, INTENT(IN), ASYN TYPE(MPI_Comm), INTENT(IN)		

MPI_Igat	herv(sendbuf, send	count, sendtype, recvbuf, recvcounts, displs,	1
	recvtype, roo	ot, comm, request, ierror) !(_c)	2
TYPE	<pre>(*), DIMENSION()</pre>	, INTENT(IN), ASYNCHRONOUS :: sendbuf	3
		'_KIND), INTENT(IN) :: sendcount	4
		TENT(IN) :: sendtype, recvtype	5
		, ASYNCHRONOUS :: recvbuf	6
		'_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)	7
		SS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)	8
	GER, INTENT(IN) ::		9
	C(MPI_Comm), INTENT		10
	-	ENT(OUT) :: request	11
TN.LF	GER, OPTIONAL, INI	'ENT(OUT) :: ierror	12 13
Fortran	binding		13 14
		COUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,	14 15
	RECVTYPE, ROO	DT, COMM, REQUEST, IERROR)	16
<typ< td=""><td>e&gt; SENDBUF(*), REC</td><td>VBUF(*)</td><td>10</td></typ<>	e> SENDBUF(*), REC	VBUF(*)	10
INTE	GER SENDCOUNT, SEN	DTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,	18
	COMM, REQUES	T, IERROR	19
Thia	coll storts a nonblock	ing variant of MDL CATHEDV (see Section 6.5)	20
1 1115	can starts a nonblock	ing variant of $MPI_GATHERV$ (see Section 6.5).	21
6101	Janhlading Coattag		22
0.12.4 1	Nonblocking Scatter		23
			24
		and and the second of the second manufacture was to second	25
MPI_ISCA	•	ount, sendtype, recvbuf, recvcount, recvtype, root, comm,	26
	request)		27
IN	sendbuf	address of send buffer (choice, significant only at	28
		$\operatorname{root})$	29
IN	sendcount	number of elements sent to each process	30
		(non-negative integer, significant only at root)	31
IN	sendtype	datatype of send buffer elements (handle, significant	32
	50	only at root)	33
OUT	recvbuf	* ,	34
	recybui	address of receive buffer (choice)	35
IN	recvcount	number of elements in receive buffer (non-negative	36
		integer)	37
IN	recvtype	datatype of receive buffer elements (handle)	38 39
IN	root	rank of sending process (integer)	40
IN	comm	communicator (handle)	41
			42
OUT	request	communication request (handle)	43
<b>a</b> 1 • • •			44
C bindin	•		45
int MPI_		d *sendbuf, int sendcount, MPI_Datatype sendtype,	46
		f, int recvcount, MPI_Datatype recvtype, int root,	47
	MP1_Comm comm	n, MPI_Request *request)	48

```
1
     int MPI_Iscatter_c(const void *sendbuf, MPI_Count sendcount,
\mathbf{2}
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
3
                   MPI_Datatype recvtype, int root, MPI_Comm comm,
4
                   MPI_Request *request)
5
     Fortran 2008 binding
6
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
7
                   root, comm, request, ierror)
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
10
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
17
                   root, comm, request, ierror) !(_c)
18
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
22
         INTEGER, INTENT(IN) :: root
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
^{24}
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
28
                   ROOT, COMM, REQUEST, IERROR)
29
         <type> SENDBUF(*), RECVBUF(*)
30
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
31
                    IERROR
32
33
         This call starts a nonblocking variant of MPI_SCATTER (see Section 6.6).
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

MPI_ISCAT	TERV(sendbuf, sendcounts, di comm, request)	spls, sendtype, recvbuf, recvcount, recvtype, root,	$\frac{1}{2}$
IN	sendbuf	address of send buffer (choice, significant only at root)	3 4 5
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)	6 7 8
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (significant only at root)	9 10 11 12
IN	sendtype	datatype of send buffer elements (handle, significant only at root)	13 14 15
OUT	recvbuf	address of receive buffer (choice)	16
IN	recvcount	number of elements in receive buffer (non-negative integer)	17 18
IN	recvtype	datatype of receive buffer elements (handle)	19
IN	root	rank of sending process (integer)	20 21
IN	comm	communicator (handle)	21
OUT	request	communication request (handle)	23
001		communication request (nancie)	24
C binding	5		25
int MPI_I		<pre>buf, const int sendcounts[],</pre>	26 27
const int displs[], MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)			
int MPI_I	scatterv_c(const void *se	ndbuf, const MPI_Count sendcounts[],	31
	const MPI_Aint displa	s[], MPI_Datatype sendtype, void *recvbuf,	32
		MPI_Datatype recvtype, int root,	33
	MPI_Comm comm, MPI_Re	equest *request)	34 35
	008 binding		36
MPI_Iscat		displs, sendtype, recvbuf, recvcount,	37
TVDE	recvtype, root, comm,	-	38
		'(IN), ASYNCHRONOUS :: sendbuf OUS :: sendcounts(*), displs(*)	39
	MPI_Datatype), INTENT(IN)	-	40
	*), DIMENSION(), ASYNCH		41 42
INTEG	ER, INTENT(IN) :: recvcou	nt, root	42
	MPI_Comm), INTENT(IN) ::		44
	MPI_Request), INTENT(OUT)	-	45
INTEG	ER, OPTIONAL, INTENT(OUT)	:: lerror	46
MPI_Iscat		displs, sendtype, recvbuf, recvcount,	47
	recvtype, root, comm,	, request, ierror) !(_c)	48

INTE INTE TYPE INTE INTE TYPE INTE Fortran MPI_ISCA	GER(KIND=MPI_COUNT_KIND), GER(KIND=MPI_ADDRESS_KIND (MPI_Datatype), INTENT(IN (*), DIMENSION(), ASYNC GER(KIND=MPI_COUNT_KIND), GER, INTENT(IN) :: root (MPI_Comm), INTENT(IN) :: (MPI_Request), INTENT(OUT GER, OPTIONAL, INTENT(OUT binding TTERV(SENDBUF, SENDCOUNTS RECVTYPE, ROOT, COMM pe> SENDBUF(*), RECVBUF(*)	<pre>HRONOUS :: recvbuf INTENT(IN) :: recvcount comm ) :: request ) :: ierror , DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 4, REQUEST, IERROR) S(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,</pre>
This	call starts a nonblocking varia	ant of MPI_SCATTERV (see Section $6.6$ ).
6125	Nonblocking Cathor to all	
0.12.5	Windlocking Gather-to-all	
MPI IALI	GATHER(sendbuf_sendcount_	sendtype recybuf recycount recytype comm
	request)	
IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative
	Schucount	integer)
IN	sendtype	datatype of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)
501		(number)
C bindir	າຍ	
	•	ndbuf, int sendcount,
	•	pe, void *recvbuf, int recvcount,
	MPI_Datatype recvtyp	pe, MPI_Comm comm, MPI_Request *request)
int MPI	Iallgather c(const void *	sendbuf, MPI_Count sendcount.
· ····	•	be, void *recvbuf, MPI_Count recvcount,
	• • • • •	be, MPI_Comm comm, MPI_Request *request)
	INTE INTE TYPE INTE TYPE INTE TYPE INTE Fortran MPI_ISCA <typ INTE China IN IN IN IN IN IN IN IN IN IN IN IN IN</typ 	INTEGER(KIND=MPI_COUNT_KIND), INTEGER(KIND=MPI_ADDRESS_KIND TYPE(MPI_Datatype), INTENT(IN) TYPE(*), DIMENSION(), ASYNC INTEGER(KIND=MPI_COUNT_KIND), INTEGER, INTENT(IN) :: root TYPE(MPI_Comm), INTENT(IN) :: TYPE(MPI_Request), INTENT(OUT INTEGER, OPTIONAL, INTENT(OUT SECUTYPE, ROOT, COM <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS COMM, REQUEST, IERR This call starts a nonblocking varia 6.12.5 Nonblocking Gather-to-all MPI_IALLGATHER(sendbuf, sendcount, request) IN sendbuf IN sendbuf IN sendcount IN sendcount IN sendtype OUT recvbuf IN recvcount IN recvtype IN comm OUT request C binding int MPI_Iallgather(const void *se MPI_Datatype sendtyp MPI_Datatype sendtyp int MPI_Iallgather_c(const void * MPI_Datatype sendtyp</type>

Fortran 2008 binding	1
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	2
comm, request, ierror)	3
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	4
INTEGER, INTENT(IN) :: sendcount, recvcount	5
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	6
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	7
TYPE(MPI_Comm), INTENT(IN) :: comm	8
TYPE(MPI_Request), INTENT(OUT) :: request	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,	11
comm, request, ierror) !(_c)	12
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf	13
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount	14
TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	15
TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	16
TYPE(MPI_Comm), INTENT(IN) :: comm	17
TYPE(MPI_Request), INTENT(OUT) :: request	18
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	19
INTEGER, OFFICIARE, INTENT(COT) TETTOT	20
Fortran binding	21
MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,	22
COMM, REQUEST, IERROR)	23
<type> SENDBUF(*), RECVBUF(*)</type>	24
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR	25
This call starts a nonblocking variant of $MPI_ALLGATHER$ (see Section 6.7).	26
This can starts a honorocking variant of with _/telo/timet( (see Section 0.1)).	27
	28
	29
	30
	31
	32
	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46

1 2	MPI_IALLO	GATHERV(sendbuf, sendcount, comm, request)	sendtype, recvbuf, recvcounts, displs, recvtype,	
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)	
5 6	IN	sendcount	number of elements in send buffer (non-negative integer)	
7	IN	sendtype	datatype of send buffer elements (handle)	
8	OUT	recvbuf	address of receive buffer (choice)	
9 10 11 12	IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	
13 14 15	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	
16 17	IN	recvtype	datatype of receive buffer elements (handle)	
18	IN	comm	communicator (handle)	
19	OUT	request	communication request (handle)	
21 22 23 24 25 26 27 28 20		allgatherv(const void *se MPI_Datatype sendtyp const int displs[], MPI_Request *request allgatherv_c(const void * MPI_Datatype sendtyp	e, void *recvbuf, const int recvcounts[], MPI_Datatype recvtype, MPI_Comm comm, ) *sendbuf, MPI_Count sendcount,	
29 30 31	Fortran 2	MPI_Datatype recvtyp	counts[], const MP1_Aint dispis[], e, MPI_Comm comm, MPI_Request *request)	
32 33 34 35 36 37 38 39 40 41 42 43	<pre>MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,</pre>			
43 44 45 46 47 48	TYPE ( INTEG TYPE (	recvtype, comm, requ	est, ierror) !(_c) F(IN), ASYNCHRONOUS :: sendbuf INTENT(IN) :: sendcount ) :: sendtype, recvtype	

INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 1 2 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN), ASYNCHRONOUS :: displs(\*) 3 TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Request), INTENT(OUT) :: request 4 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 5 6 Fortran binding MPI\_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, 8 RECVTYPE, COMM, REQUEST, IERROR) 9 <type> SENDBUF(\*), RECVBUF(\*) 10 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 11 REQUEST, IERROR 1213 This call starts a nonblocking variant of MPI\_ALLGATHERV (see Section 6.7). 14156.12.6 Nonblocking All-to-All Scatter/Gather 161718 MPI\_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request) 19 20IN sendbuf starting address of send buffer (choice) 2122 IN sendcount number of elements sent to each process 23(non-negative integer) 24IN sendtype datatype of send buffer elements (handle) 25OUT recvbuf address of receive buffer (choice) 2627IN number of elements received from any process recvcount 28 (non-negative integer) 29 IN recvtype datatype of receive buffer elements (handle) 30 IN communicator (handle) comm 3132 OUT communication request (handle) request 33 34 C binding 35 int MPI\_Ialltoall(const void \*sendbuf, int sendcount, 36 MPI\_Datatype sendtype, void \*recvbuf, int recvcount, 37 MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request \*request) 38 int MPI\_Ialltoall\_c(const void \*sendbuf, MPI\_Count sendcount, 39 MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 40 MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Request \*request) 41 42Fortran 2008 binding 43 MPI\_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, 44 comm, request, ierror) 45TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf 46INTEGER, INTENT(IN) :: sendcount, recvcount 47TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 48

1	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf
2	TYPE(MPI_Comm), INTENT(IN) :: comm
3	TYPE(MPI_Request), INTENT(OUT) :: request
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5 6	MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
7	<pre>comm, request, ierror) !(_c)</pre>
8	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf
9	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
10	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
11	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf
12	TYPE(MPI_Comm), INTENT(IN) :: comm
13	TYPE(MPI_Request), INTENT(OUT) :: request
14	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15	Fortran binding
16	MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
17	COMM, REQUEST, IERROR)
18 19	<type> SENDBUF(*), RECVBUF(*)</type>
20	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
21	This call starts a nonblocking variant of $MPI_ALLTOALL$ (see Section 6.8).
22	
23	
24	
25	
26	
27 28	
29	
30	
31	
32	
33	
34	
35	
36 37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

MPI_IALI	TOALLV(sendbuf, sendcour_ recvtype, comm, requ	nts, sdispls, sendtype, recvbuf, recvcounts, rdispls, uest)	1 2
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	non-negative integer array (of length group size)	4 5
		specifying the number of elements to send to each	6
		rank	7
IN	sdispls	integer array (of length group size). Entry <b>j</b> specifies	8
		the displacement (relative to sendbuf) from which to take the outgoing data destined for process <b>j</b>	9 10
IN	sendtype	datatype of send buffer elements (handle)	11 12
OUT	recvbuf	address of receive buffer (choice)	13
IN	recvcounts	non-negative integer array (of length group size)	14
		specifying the number of elements that can be	15
		received from each rank	16 17
IN	rdispls	integer array (of length group size). Entry i specifies	18
		the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from process i	19
IN	requiture		20
	recvtype	datatype of receive buffer elements (handle)	21 22
IN	comm	communicator (handle)	22
OUT	request	communication request (handle)	24
C bindiı	nor		25
	0	<pre>*sendbuf, const int sendcounts[],</pre>	26 27
	const int sdispla	s[], MPI_Datatype sendtype, void *recvbuf,	21
		<pre>unts[], const int rdispls[],</pre>	29
	MPI_Datatype recv	type, MPI_Comm comm, MPI_Request *request)	30
int MPI_		d *sendbuf, const MPI_Count sendcounts[],	31
		<pre>displs[], MPI_Datatype sendtype,</pre>	32 33
		onst MPI_Count recvcounts[], Hispls[], MPI_Datatype recvtype,	34
		PI_Request *request)	35
Fontnon			36
	2008 binding toally(sendbuf, sendco	unts, sdispls, sendtype, recvbuf, recvcounts,	37
		e, comm, request, ierror)	38 39
		TENT(IN), ASYNCHRONOUS :: sendbuf	40
INTE		<pre>HRONOUS :: sendcounts(*), sdispls(*),</pre>	41
тург	recvcounts(*), r	dispis(*) (IN) :: sendtype, recvtype	42
	E(*), DIMENSION(), AS		43 44
	E(MPI_Comm), INTENT(IN)		44 45
	E(MPI_Request), INTENT(	-	46
INTE	EGER, OPTIONAL, INTENT(	OUT) :: ierror	47

```
1
     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
\mathbf{2}
                    rdispls, recvtype, comm, request, ierror) !(_c)
3
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                    sendcounts(*), recvcounts(*)
5
6
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
7
                    rdispls(*)
8
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
9
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     Fortran binding
14
     MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
15
                    RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
16
         <type> SENDBUF(*), RECVBUF(*)
17
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
18
                    RECVTYPE, COMM, REQUEST, IERROR
19
20
         This call starts a nonblocking variant of MPI_ALLTOALLV (see Section 6.8).
21
22
23
^{24}
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_IALL	TOALLW(sendbuf, sendcour recvtypes, comm, req	nts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, uest)	1 $2$
IN	sendbuf	starting address of send buffer (choice)	3
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)	4 5 6 7
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)	8 9 10 11
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)	12 13 14 15
OUT	recvbuf	address of receive buffer (choice)	16
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)	17 18 19
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)	20 21 22 23 24
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)	25 26 27
IN	comm	communicator (handle)	28
OUT	request	communication request (handle)	29 30 31
C bindin	0		32
int MPI		<pre>*sendbuf, const int sendcounts[], [], const MPI_Datatype sendtypes[],</pre>	33
	-	<pre>nst int recvcounts[], const int rdispls[],</pre>	$\frac{34}{35}$
	const MPI_Datatyp	e recvtypes[], MPI_Comm comm,	36
	MPI_Request *requ	est)	37
int MPI_	[alltoallw_c(const void	d *sendbuf, const MPI_Count sendcounts[],	38
		<pre>ispls[], const MPI_Datatype sendtypes[],</pre>	39 40
		<pre>nst MPI_Count recvcounts[], ianla[] const MPI_Datatuma recutumes[]</pre>	40 41
		<pre>ispls[], const MPI_Datatype recvtypes[], I_Request *request)</pre>	42
			43
Fortran 2008 binding			44

MPI\_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,

TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf

recvcounts, rdispls, recvtypes, comm, request, ierror)

45

46

```
1
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
\mathbf{2}
                    recvcounts(*), rdispls(*)
3
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
4
                    recvtypes(*)
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
         TYPE(MPI_Comm), INTENT(IN) :: comm
7
         TYPE(MPI_Request), INTENT(OUT) :: request
8
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9
     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
10
                   recvcounts, rdispls, recvtypes, comm, request, ierror) !(_c)
11
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
12
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
13
                    sendcounts(*), recvcounts(*)
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
15
                    rdispls(*)
16
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
17
                    recvtypes(*)
18
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     Fortran binding
24
     MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
25
                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
26
         <type> SENDBUF(*), RECVBUF(*)
27
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
28
                    RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
29
         This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 6.8).
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

# 6.12.7 Nonblocking Reduce

	0		2
MPI IRF	DUCE(sendbuf_recybuf	count, datatype, op, root, comm, request)	3 4
IN	sendbuf	address of send buffer (choice)	5
OUT	recvbuf	address of receive buffer (choice, significant only at root)	6 7 8
IN	count	number of elements in send buffer (non-negative integer)	9 10
IN	datatype	datatype of elements of send buffer (handle)	11 12
IN	ор	reduce operation (handle)	13
IN	root	rank of root process (integer)	14
IN	comm	communicator (handle)	15
OUT	request	communication request (handle)	16 17
001	loquose	communication request (namalo)	18
C bindi int MPI	_Ireduce(const void	<pre>*sendbuf, void *recvbuf, int count, datatype, MPI_Op op, int root, MPI_Comm comm, request)</pre>	19 20 21 22 23
int MPI		id *sendbuf, void *recvbuf, MPI_Count count, datatype, MPI_Op op, int root, MPI_Comm comm, request)	24 25 26
MPI_Ire TYP TYP INT TYP TYP TYP TYP	<pre>ierror) E(*), DIMENSION(); E(*), DIMENSION(); EGER, INTENT(IN) ::</pre>	TENT(IN) :: datatype N) :: op (IN) :: comm ENT(OUT) :: request	27 28 29 30 31 32 33 34 35 36 37 38
TYP TYP INT TYP	ierror) !(_c) E(*), DIMENSION() E(*), DIMENSION() EGER(KIND=MPI_COUNT E(MPI_Datatype), INT	uf, count, datatype, op, root, comm, request, , INTENT(IN), ASYNCHRONOUS :: sendbuf , ASYNCHRONOUS :: recvbuf _KIND), INTENT(IN) :: count TENT(IN) :: datatype	38 39 40 41 42 43 44
	E(MPI_Op), INTENT(IN EGER, INTENT(IN) ::	-	45 46
	E(MPI_Comm), INTENT		40 47
TYP	E(MPI_Request), INTH	ENT(OUT) :: request	48

1 2	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
3	Fortran	binding			
4	MPI_IRED	UCE(SENDBUF, RECVBUE	F, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,		
5	IERROR)				
6	<type> SENDBUF(*), RECVBUF(*)</type>				
7	INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR				
8 9	This call starts a nonblocking variant of $MPI_REDUCE$ (see Section 6.9.1).				
10	Adi	vice to implementors.	The implementation is explicitly allowed to use different		
11	algo	orithms for blocking and	I nonblocking reduction operations that might change the		
12	orde	er of evaluation of the o	operations. However, as for MPI_REDUCE, it is strongly		
13			DUCE be implemented so that the same result be obtained		
14			plied on the same arguments, appearing in the same order.		
15			optimizations that take advantage of the physical location		
16	or p	rocesses. (End of advice	e to implementors.)		
17	Adi	vice to users. For opera	ations which are not truly associative, the result delivered		
18 19	upo	n completion of the non	blocking reduction may not exactly equal the result deliv-		
20			ion, even when specifying the same arguments in the same		
21	orde	er. (End of advice to us	ers.)		
22	C 10 0 M				
23	6.12.8 I	Nonblocking All-Reduce			
24					
25 26	MPI IALI	REDUCE(sendbuf, recyc	ouf, count, datatype, op, comm, request)		
20	IN	sendbuf			
28			starting address of send buffer (choice)		
29	OUT	recvbuf	starting address of receive buffer (choice)		
30 31	IN	count	number of elements in send buffer (non-negative integer)		
32	IN	datatype	datatype of elements of send buffer (handle)		
33	IN	ор	operation (handle)		
34 35	IN	comm	communicator (handle)		
36	OUT	request	communication request (handle)		
37		•			
38	C bindir	ıg			
39	int MPI_	Iallreduce(const voi	id *sendbuf, void *recvbuf, int count,		
40		MPI_Datatype d	atatype, MPI_Op op, MPI_Comm comm,		
41		MPI_Request *r	equest)		
42	int MPT	Iallreduce c(const v	void *sendbuf, void *recvbuf, MPI_Count count,		
43	<b></b> -		atatype, MPI_Op op, MPI_Comm comm,		
44 45		MPI_Request *r			
45	Fortran	2008 hinding			
47	Fortran 2008 binding MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,				
48		ierror)	, count, adoatjpo, op, comm, requebt,		
		/			

TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 1  $\mathbf{2}$ TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 3 INTEGER, INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype 4 TYPE(MPI\_Op), INTENT(IN) :: op 5TYPE(MPI\_Comm), INTENT(IN) :: comm 6 7 TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 9 MPI\_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request, 10 ierror) !(\_c) 11 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 12TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 13 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 14TYPE(MPI\_Datatype), INTENT(IN) :: datatype 15TYPE(MPI\_Op), INTENT(IN) :: op 16TYPE(MPI\_Comm), INTENT(IN) :: comm 17 TYPE(MPI\_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 Fortran binding 2021MPI\_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, 22 IERROR) 23<type> SENDBUF(\*), RECVBUF(\*) 24INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 25This call starts a nonblocking variant of MPI\_ALLREDUCE (see Section 6.9.6). 26276.12.9 Nonblocking Reduce-Scatter with Equal Blocks 2829 30 MPI\_IREDUCE\_SCATTER\_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm, 31request) 32 33 sendbuf IN starting address of send buffer (choice) 34 OUT recvbuf starting address of receive buffer (choice) 35 IN recvcount element count per block (non-negative integer) 36 37 IN datatype datatype of elements of send and receive buffers 38 (handle) 39 IN op operation (handle) 40 41 IN communicator (handle) comm 42OUT communication request (handle) request 43 44C binding 45int MPI\_Ireduce\_scatter\_block(const void \*sendbuf, void \*recvbuf, 46int recvcount, MPI\_Datatype datatype, MPI\_Op op, 47

MPI\_Comm comm, MPI\_Request \*request)

```
1
     int MPI_Ireduce_scatter_block_c(const void *sendbuf, void *recvbuf,
\mathbf{2}
                   MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
3
                   MPI_Comm comm, MPI_Request *request)
4
     Fortran 2008 binding
5
     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
6
                   request, ierror)
7
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
9
         INTEGER, INTENT(IN) :: recvcount
10
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
         TYPE(MPI_Op), INTENT(IN) :: op
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
17
                   request, ierror) !(_c)
18
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Op), INTENT(IN) :: op
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
^{24}
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_IREDUCE_SCATTER_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
28
                   REQUEST, IERROR)
29
         <type> SENDBUF(*), RECVBUF(*)
30
         INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
^{31}
32
         This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Sec-
33
     tion 6.10.1).
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

# 6.12.10 Nonblocking Reduce-Scatter

	0		2
			3 4
MPI_IR	EDUCE_SCATTER(sendbi	uf, recvbuf, recvcounts, datatype, op, comm, request)	5
IN	sendbuf	starting address of send buffer (choice)	6
OUT	recvbuf	starting address of receive buffer (choice)	7
IN	recvcounts	non-negative integer array specifying the number of elements in result distributed to each process. This array must be identical on all calling processes.	8 9 10
IN	datatype	datatype of elements of input buffer (handle)	11 12
IN	ор	operation (handle)	13
IN	comm	communicator (handle)	14
OUT			15
001	request	communication request (handle)	16
C bind	ing		17
	0	st void *sendbuf, void *recvbuf,	18 19
		counts[], MPI_Datatype datatype, MPI_Op op,	20
		MPI_Request *request)	21
int MD	Traduca scatter c(c	onst void *sendbuf, void *recvbuf,	22
IIIC MI		t recvcounts[], MPI_Datatype datatype,	23
		_Comm comm, MPI_Request *request)	24
The data			25
	n 2008 binding	, recvbuf, recvcounts, datatype, op, comm,	26
MFI_II(	request, ierro		27 28
TYI	-	INTENT(IN), ASYNCHRONOUS :: sendbuf	29
		ASYNCHRONOUS :: recvbuf	30
INT	TEGER, INTENT(IN), ASY	YNCHRONOUS :: recvcounts(*)	31
	<pre>PE(MPI_Datatype), INTH</pre>	• •	32
	PE(MPI_Op), INTENT(IN)	-	33
	PE(MPI_Comm), INTENT(I		34
	PE(MPI_Request), INTEN		35
1N.	TEGER, OPTIONAL, INTEN	VI(UUI) :: lerror	36 37
MPI_Ire		, recvbuf, recvcounts, datatype, op, comm,	38
	request, ierro		39
		INTENT(IN), ASYNCHRONOUS :: sendbuf	40
		ASYNCHRONOUS :: recvbuf	41
	PE(MPI_Datatype), INT	<pre>XIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*) </pre>	42
	PE(MPI_Datatype), INT PE(MPI_Op), INTENT(IN)	V-1	43
	PE(MPI_Comm), INTENT(1	-	44
	PE(MPI_Request), INTEN		45
	TEGER, OPTIONAL, INTEN	-	46 47
			47 48
			10

1	The state of the	1 • 1•	
2	Fortran binding MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,		
3	REQUEST, IERROR)		
4	<typ< td=""><td>e&gt; SENDBUF(*), RECVBUF(*)</td><td></td></typ<>	e> SENDBUF(*), RECVBUF(*)	
5	INTE	GER RECVCOUNTS(*), DATATY	PE, OP, COMM, REQUEST, IERROR
6	This	call starts a nonblocking varia	nt of MPI_REDUCE_SCATTER (see Section 6.10.2).
7 8			
9	6.12.11	Nonblocking Inclusive Scan	
10			
11			
12	MPI_ISCA	N(sendbuf, recvbuf, count, data	atype, op, comm, request)
13 14	IN	sendbuf	starting address of send buffer (choice)
15	OUT	recvbuf	starting address of receive buffer (choice)
16 17	IN	count	number of elements in input buffer (non-negative integer)
18	IN	datatype	datatype of elements of input buffer (handle)
19 20	IN	ор	operation (handle)
21	IN	comm	communicator (handle)
22	OUT	request	communication request (handle)
23 24			
25	C bindin	0	
26	int MPI_		, void *recvbuf, int count,
27		MPI_Datatype datatyp MPI_Request *request	e, MPI_Op op, MPI_Comm comm,
28			
29 30	int MPL_		uf, void *recvbuf, MPI_Count count,
31		MPI_Request *request	e, MPI_Op op, MPI_Comm comm, )
32	<b>D</b> (		, ,
33		2008 binding	, datatype, op, comm, request, ierror)
34			T(IN), ASYNCHRONOUS :: sendbuf
35 36		(*), DIMENSION(), ASYNC	
37	INTE	GER, INTENT(IN) :: count	
38		(MPI_Datatype), INTENT(IN	
39		(MPI_Op), INTENT(IN) :: o	-
40		(MPI_Comm), INTENT(IN) ::	
41		(MPI_Request), INTENT(OUT GER, OPTIONAL, INTENT(OUT	-
42			
43 44	MPI_Isca		, datatype, op, comm, request, ierror)
45	TYPF	!(_c) (*) DIMENSION() INTEN	T(IN), ASYNCHRONOUS :: sendbuf
46		(*), DIMENSION(), INTEN (*), DIMENSION(), ASYNC	
47	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count		
48	TYPE	(MPI_Datatype), INTENT(IN	) :: datatype

TYP TYP	-	1		
Fortran MPI_ISC <ty INT</ty 	binding AN(SENDBUF, RECVB pe> SENDBUF(*), RI EGER COUNT, DATAT	5 UF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) ECVBUF(*) YPE, OP, COMM, REQUEST, IERROR cking variant of MPI_SCAN (see Section 6.11).		
6.12.12	Nonblocking Exclu	sive Scan		
MPI_IEX	SCAN(sendbuf, recvb	uf, count, datatype, op, comm, request) 15		
IN	sendbuf	starting address of send buffer (choice) 17		
OUT	recvbuf	starting address of receive buffer (choice) $18$		
IN	count	number of elements in input buffer (non-negative integer) 21		
IN	datatype	datatype of elements of input buffer (handle) 22		
IN	ор	operation (handle) <sup>23</sup>		
IN	comm	intra-communicator (handle) 24 25		
OUT	request	communication request (handle) 26		
C bindi int MPI	_Iexscan(const vo	27 28 id *sendbuf, void *recvbuf, int count, e datatype, MPI_Op op, MPI_Comm comm, *request)		
int MPI	_Iexscan_c(const	void *sendbuf, void *recvbuf, MPI_Count count, a33 e datatype, MPI_Op op, MPI_Comm comm, 34		
Fortran 2008 binding MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf INTEGER, INTENT(IN) :: count TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Comm), INTENT(IN) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40				
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror) !(_c)				

1	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf
2	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf
3	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
4	TYPE(MPI_Datatype), INTENT(IN) :: datatype
5	TYPE(MPI_Op), INTENT(IN) :: op
6	TYPE(MPI_Comm), INTENT(IN) :: comm
7	TYPE(MPI_Request), INTENT(OUT) :: request
8	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9	
10	Fortran binding
11	MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
12	<type> SENDBUF(*), RECVBUF(*)</type>
13	INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
14	This call starts a nonblocking variant of $MPI_EXSCAN$ (see Section 6.11.2).

6.13 Persistent Collective Operations

18 Many parallel computation algorithms involve repetitively executing a collective commu-19nication operation with the same arguments each time. As with persistent point-to-point 20operations (see Section 3.9), persistent collective operations allow the MPI programmer to 21specify operations that will be reused frequently (with fixed arguments). MPI can be de-22signed to select a more efficient way to perform the collective operation based on the param-23eters specified when the operation is initialized. This "planned-transfer" approach [52, 41]  $^{24}$ can offer significant performance benefits for programs with repetitive communication pat-25terns. 26

In terms of data movement, each persistent collective operation has the same effect as its blocking and nonblocking counterparts for intra-communicators and inter-communicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 6.9.1).

Initialization calls for MPI persistent collective operations are non-local and follow all the existing rules for collective operations, in particular ordering; programs that do not conform to these restrictions are erroneous. After initialization, all arrays associated with input arguments (such as arrays of counts, displacements, and datatypes in the vector versions of the collectives) must not be modified until the corresponding persistent request is freed with MPI\_REQUEST\_FREE.

According to the definitions in Section 2.4.2, the persistent collective initialization procedures are incomplete. They are also non-local procedures because they may or may not return before they are called in all MPI processes of the process group associated with the specified communicator.

Advice to users. This is one of the exceptions in which incomplete procedures are non-local and therefore blocking. (End of advice to users.)

The request argument is an output argument that can be used zero or more times with MPI\_START or MPI\_STARTALL in order to start the collective operation. The request is initially inactive after the initialization call. Once initialized, persistent collective operations can be started in any order and the order can differ among processes in the communicator.

15 16

17

42

43

*Rationale.* All ordering requirements that an implementation may need to match up collective operations across the communicator are achieved through the ordering requirements of the initialization functions. This enables out-of-order starts for the persistent operations, and particularly supports their use in MPI\_STARTALL. (*End of rationale.*)

Advice to implementors. An MPI implementation should do no worse than duplicating the communicator during the initialization function, caching the input arguments, and calling the appropriate nonblocking collective function, using the cached arguments, during MPI\_START. High-quality implementations should be able to amortize setup costs and further optimize by taking advantage of early-binding, such as efficient and effective pre-allocation of certain resources and algorithm selection. (*End* of advice to implementors.)

A request must be inactive when it is started. Starting the operation makes the request active. Once any process starts a persistent collective operation, it must complete that operation and all other processes in the communicator must eventually start (and complete) the same persistent collective operation. Persistent collective operations cannot be matched with blocking or nonblocking collective operations. Completion of a persistent collective operation makes the corresponding request inactive. After starting a persistent collective operation, all associated send buffers must not be modified and all associated receive buffers must not be accessed until the corresponding persistent request is completed.

Completing a persistent collective request, for example using MPI\_TEST or MPI\_WAIT, makes it inactive, but does not free the request. This is the same behavior as for persistent point-to-point requests. Inactive persistent collective requests can be freed using MPI\_REQUEST\_FREE. It is erroneous to free an active persistent collective request. Persistent collective operations cannot be canceled; it is erroneous to use MPI\_CANCEL on a persistent collective request.

For every nonblocking collective communication operation in MPI, there is a corresponding persistent collective operation with the analogous API signature.

The collective persistent API signatures include an info object in order to support optimization hints and other information that may be nonstandard. Persistent collective operations may be optimized during communicator creation or by the initialization operation of an individual persistent collective. Note that communicator-scoped hints should be provided using MPI\_COMM\_SET\_INFO while, for operation-scoped hints, they are supplied to the persistent collective communication initialization functions using the info argument.

## 6.13.1 Persistent Barrier Synchronization

MPI\_BARRIER\_INIT(comm, info, request)

IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

### C binding

int MPI\_Barrier\_init(MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request)

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13 14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36 37

38 39 40

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Barrier_init(comm, info, request, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Info), INTENT(IN) :: info
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     Fortran binding
8
     MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
9
         INTEGER COMM, INFO, REQUEST, IERROR
10
11
         Creates a persistent collective communication request for the barrier operation.
12
13
     6.13.2 Persistent Broadcast
14
15
16
     MPI_BCAST_INIT(buffer, count, datatype, root, comm, info, request)
17
       INOUT
                buffer
                                            starting address of buffer (choice)
18
19
       IN
                                            number of entries in buffer (non-negative integer)
                count
20
                                            datatype of buffer (handle)
       IN
                datatype
21
       IN
                root
                                            rank of broadcast root (integer)
22
23
       IN
                comm
                                            communicator (handle)
24
       IN
                info
                                            info argument (handle)
25
       OUT
                request
                                            communication request (handle)
26
27
28
     C binding
     int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype,
29
30
                    int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
^{31}
     int MPI_Bcast_init_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
32
                    int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
33
34
     Fortran 2008 binding
35
     MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
37
         INTEGER, INTENT(IN) :: count, root
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Info), INTENT(IN) :: info
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
44
                    !(_c)
45
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

TYPE TYPE TYPE	GER, INTENT(IN) :: roo (MPI_Comm), INTENT(IN) (MPI_Info), INTENT(IN) (MPI_Request), INTENT( GER, OPTIONAL, INTENT(	:: comm :: info OUT) :: request	1 2 3 4 5
<type INTE</type 	I_INIT(BUFFER, COUNT, ∋> BUFFER(*) GER COUNT, DATATYPE, R	DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR) OOT, COMM, INFO, REQUEST, IERROR ommunication request for the broadcast operation.	6 7 9 10 11
6.13.3 P	ersistent Gather		12 13 14 15
MPI_GAT	HER_INIT(sendbuf, sendco info, request)	unt, sendtype, recvbuf, recvcount, recvtype, root, comm,	16 17 18
IN	sendbuf	starting address of send buffer (choice)	19
IN	sendcount	number of elements in send buffer (non-negative integer)	20 21
IN	sendtype	datatype of send buffer elements (handle)	22 23
OUT	recvbuf	address of receive buffer (choice, significant only at root)	24 25
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)	26 27
IN	recvtype	datatype of recv buffer elements (handle, significant only at root)	28 29 30
IN	root	rank of receiving process (integer)	31
IN	comm	communicator (handle)	32
IN	info	info argument (handle)	33
OUT	request	communication request (handle)	34 35
	·		36
C bindin	g		37
int MPI_(		*sendbuf, int sendcount,	38
	• •	<pre>ltype, void *recvbuf, int recvcount, rtype, int root, MPI_Comm comm, MPI_Info info,</pre>	39 40
	MPI_Request *requ		41
int MDT (			42
IIIC MPI_(		<pre>id *sendbuf, MPI_Count sendcount, ltype, void *recvbuf, MPI_Count recvcount,</pre>	43
		vtype, int root, MPI_Comm comm, MPI_Info info,	44 45
	MPI_Request *request)		45 46
			47
			48

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
3
                   root, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         TYPE(MPI_Info), INTENT(IN) :: info
10
         TYPE(MPI_Request), INTENT(OUT) :: request
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
13
                   root, comm, info, request, ierror) !(_c)
14
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
15
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
16
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
18
         INTEGER, INTENT(IN) :: root
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Info), INTENT(IN) :: info
21
         TYPE(MPI_Request), INTENT(OUT) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
^{24}
     Fortran binding
25
     MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
26
                   ROOT, COMM, INFO, REQUEST, IERROR)
27
         <type> SENDBUF(*), RECVBUF(*)
28
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
29
                    REQUEST, IERROR
30
         Creates a persistent collective communication request for the gather operation.
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

		$\frac{1}{2}$	
sendbuf	starting address of send buffer (choice)	3	
sendcount	number of elements in send buffer (non-negative integer)	4 5 6	
sendtype	<i>c</i> ,	7	
recvbuf	address of receive buffer (choice, significant only at root)	8 9	
recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)	10 11 12 13	
displs	integer array (of length group size). Entry i specifies the displacement relative to <b>recvbuf</b> at which to place the incoming data from process i (significant only at root)	14 15 16 17	
recvtype	datatype of recv buffer elements (handle, significant only at root)	18 19 20	
root	rank of receiving process (integer)	20	
comm	communicator (handle)	22	
info	info argument (handle)	23	
request	communication request (handle)	24 25	
	- ( )	26	
		27	
MPI_Datatype sendtype const int displs[], M	e, void *recvbuf, const int recvcounts[], MPI_Datatype recvtype, int root,	28 29 30 31	
atherv_init_c(const void MPI_Datatype sendtype const MPI_Count recvo MPI_Datatype recvtype	<pre>*sendbuf, MPI_Count sendcount, e, void *recvbuf, counts[], const MPI_Aint displs[], e, int root, MPI_Comm comm, MPI_Info info,</pre>	32 33 34 35 36 37	
Fortran 2008 binding38MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, info, request, ierror)39TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf40INTEGER, INTENT(IN) :: sendcount, root42TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype43TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf44INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)45TYPE(MPI_Comm), INTENT(IN) :: comm46TYPE(MPI_Info), INTENT(IN) :: info47			
	<pre>root, comm, info, request) sendbuf sendcount sendtype recvbuf recvcounts displs recvtype root comm info request atherv_init(const void *s     MPI_Datatype sendtype     const int displs[], M MPI_Comm comm, MPI_Ir atherv_init_c(const void     MPI_Datatype sendtype     const int displs[], M MPI_Comm comm, MPI_Ir atherv_init_c(const void     MPI_Datatype recvtype     MPI_Datatype recvtype     MPI_Request *request) 008 binding rv_init(sendbuf, sendcoun     recvtype, root, comm, *), DIMENSION(), INTENT ER, INTENT(IN) :: sendcoun MPI_Comm), INTENT(IN) :: MPI_Info), INTENT(IN) :: MPI_Info), INTENT(IN) :: </pre>	<pre>sendcount number of elements in send buffer (non-negative integer) sendtype datatype of send buffer elements (handle) address of receive buffer (choice, significant only at root) recvcounts non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root) displs integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root) recvtype datatype of recv buffer elements (handle, significant only at root) root rank of receiving process (integer) comm communicator (handle) info info argument (handle) request communication request (handle) atherv_init(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request) atherv_init_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint displs[], MPI_Datatype recvtype, int root, MPI_Info info, MPI_Request *request) 008 binding rv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, info, request, ierror) *), DIMENSION(), INTENT(IN) :: sendtype, recvtype *), DIMENSION(), ASYNCHRONOUS :: recvbuf ER, INTENT(IN) :: sendtype, recvtype *), DIMENSION(), ASYNCHRONOUS :: recvbuf ER, INTENT(IN) :: comm</pre>	

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
3
                   recvtype, root, comm, info, request, ierror) !(_c)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
10
         INTEGER, INTENT(IN) :: root
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Info), INTENT(IN) :: info
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
     MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
18
                   RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
19
         <type> SENDBUF(*), RECVBUF(*)
20
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
21
                    COMM, INFO, REQUEST, IERROR
22
         Creates a persistent collective communication request for the gathery operation.
23
^{24}
25
26
27
28
29
30
^{31}
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

## 6.13.4 Persistent Scatter

MPI_SCATTER_INIT(sendbuf, sendcount,	, sendtype,	recvbuf,	recvcount,	recvtype,	root,
comm, info, request)					

			6
IN	sendbuf	address of send buffer (choice, significant only at	7
		root)	8
IN	sendcount	number of elements sent to each process	9
		(non-negative integer, significant only at root)	10
IN	sendtype	datatype of send buffer elements (handle, significant	11
		only at root)	12
- · ·		• ,	13
OUT	recvbuf	address of receive buffer (choice)	14
IN	recvcount	number of elements in receive buffer (non-negative	15
		integer)	16
IN	recvtype	datatype of receive buffer elements (handle)	17
		* <b>-</b>	18
IN	root	rank of sending process (integer)	19
IN	comm	communicator (handle)	20
IN	info	info argument (handle)	21
		$\ddot{\mathbf{Q}}$	22
OUT	request	communication request (handle)	23

# C binding

<pre>int MPI_Scatter_init(const void *sendbuf, int sendcount,</pre>	26
<pre>MPI_Datatype sendtype, void *recvbuf, int recvcount,</pre>	27
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	28
MPI_Request *request)	29
<pre>int MPI_Scatter_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	30
Int Mri_Scatter_Init_c(const Void *Senabai, Mri_count Senacount,	31

MPI_Datatype sendtype,	<pre>void *recvbuf, MPI_Count recvcount,</pre>
MPI_Datatype recvtype,	<pre>int root, MPI_Comm comm, MPI_Info info,</pre>
MPI_Request *request)	

#### Fortran 2008 binding MPI\_Scatter\_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request, ierror) TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf INTEGER, INTENT(IN) :: sendcount, recvcount, root TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf TYPE(MPI\_Comm), INTENT(IN) :: comm TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Request), INTENT(OUT) :: request INTEGER, OPTIONAL, INTENT(OUT) :: ierror MPI\_Scatter\_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request, ierror) !(\_c)

1 2 3 4 5 6 7 8 9 10	INTE TYPE TYPE INTE TYPE TYPE INTE	GER(KIND=MPI_COUN (MPI_Datatype), I (*), DIMENSION( GER, INTENT(IN) : (MPI_Comm), INTEN (MPI_Info), INTEN (MPI_Request), IN GER, OPTIONAL, IN binding	T(IN) :: comm T(IN) :: info TENT(OUT) :: request TENT(OUT) :: ierror	
12	MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)			
13	<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>			
14 15	INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,			
16	REQUEST, IERROR			
17	Creates a persistent collective communication request for the scatter operation.			
18				
19 20	MPI_SCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype,			
20	root, comm, info, request)			
22	IN	sendbuf	address of send buffer (choice, significant only at	
23			root)	
24 25 26	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank (significant only at root)	
27 28 29 30 31	IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i (significant only at root)	
32 33	IN	sendtype	datatype of send buffer elements (handle, significant only at root)	
34 35	OUT	recvbuf	address of receive buffer (choice, significant only at root)	
36 37 38	IN	recvcount	number of elements in receive buffer (non-negative integer)	
39	IN	recvtype	datatype of receive buffer elements (handle)	
40	IN	root	rank of sending process (integer)	
41	IN	comm	communicator (handle)	
42 43	IN	info	info argument (handle)	
43 44	OUT	request	communication request (handle)	
45		•	•	
46	C binding			
47	<pre>int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],</pre>			
<pre>48 const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>				

```
1
              int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
                                                                                   2
              MPI_Info info, MPI_Request *request)
                                                                                   3
int MPI_Scatterv_init_c(const void *sendbuf, const MPI_Count sendcounts[],
              const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf,
                                                                                   5
              MPI_Count recvcount, MPI_Datatype recvtype, int root,
                                                                                   6
              MPI_Comm comm, MPI_Info info, MPI_Request *request)
                                                                                   7
                                                                                   8
Fortran 2008 binding
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                                                                   9
                                                                                   10
              recvcount, recvtype, root, comm, info, request, ierror)
                                                                                   11
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
                                                                                   12
                                                                                   13
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   14
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   15
    INTEGER, INTENT(IN) :: recvcount, root
                                                                                   16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   17
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                                                                   21
              recvcount, recvtype, root, comm, info, request, ierror) !(_c)
                                                                                   22
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   23
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
                                                                                   24
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                   25
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   27
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                   28
    INTEGER, INTENT(IN) :: root
                                                                                   29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   30
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   31
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   33
                                                                                   34
Fortran binding
                                                                                   35
MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
                                                                                   36
              RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
                                                                                   37
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   38
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
                                                                                   39
               COMM, INFO, REQUEST, IERROR
                                                                                   40
    Creates a persistent collective communication request for the scattery operation.
                                                                                   41
                                                                                   42
                                                                                   43
```

```
286
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.13.5 Persistent Gather-to-all
\mathbf{2}
3
4
     MPI_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recybuf, recycount, recytype, comm,
5
                    info, request)
6
       IN
                sendbuf
                                            starting address of send buffer (choice)
7
       IN
8
                sendcount
                                            number of elements in send buffer (non-negative
9
                                            integer)
10
       IN
                sendtype
                                            datatype of send buffer elements (handle)
11
       OUT
                recvbuf
                                            address of receive buffer (choice)
12
       IN
                                            number of elements received from any process
13
                 recvcount
14
                                            (non-negative integer)
15
       IN
                recvtype
                                            datatype of receive buffer elements (handle)
16
       IN
                comm
                                            communicator (handle)
17
       IN
                info
18
                                            info argument (handle)
19
       OUT
                request
                                            communication request (handle)
20
21
     C binding
22
     int MPI_Allgather_init(const void *sendbuf, int sendcount,
23
                    MPI_Datatype sendtype, void *recvbuf, int recvcount,
24
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
25
                    MPI_Request *request)
26
27
     int MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcount,
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
28
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
29
30
                    MPI_Request *request)
^{31}
     Fortran 2008 binding
32
     MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
33
                    recvtype, comm, info, request, ierror)
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
         INTEGER, INTENT(IN) :: sendcount, recvcount
36
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Info), INTENT(IN) :: info
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
44
                    recvtype, comm, info, request, ierror) !(_c)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
47
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

	TYPE(MPI_Comm), INTENT(IN) :		1
	<pre>SYPE(MPI_Info), INTENT(IN) :</pre>		2 3
	TYPE(MPI_Request), INTENT(OU INTEGER, OPTIONAL, INTENT(OU	-	3
_	INTEGER, OPTIONAL, INTENI(OU	I) :: leffor	5
	ran binding		6
MPI_A		COUNT, SENDTYPE, RECVBUF, RECVCOUNT,	7
	<pre> RECVITE, COMM, INF  type&gt; SENDBUF(*), RECVBUF(*)</pre>	O, REQUEST, IERROR)	8
		RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	9 10 11
(	Creates a persistent collective con	munication request for the allgather operation.	12 13
MPI_	ALLGATHERV_INIT(sendbuf, send comm, info, request)	dcount, sendtype, recvbuf, recvcounts, displs, recvtype,	14 15 16
IN	sendbuf	starting address of send buffer (choice)	17
IN	sendcount	number of elements in send buffer (non-negative integer)	18 19 20
IN	sendtype	datatype of send buffer elements (handle)	21
OU <sup>.</sup>	T recvbuf	address of receive buffer (choice)	22
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process	23 24 25 26
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i	27 28 29
IN	recvtype	datatype of receive buffer elements (handle)	30
IN	comm	communicator (handle)	31 32
IN	info	info argument (handle)	33
OU.	T request	communication request (handle)	34
			35
C bi	nding		36
int N	<pre>IPI_Allgatherv_init(const vo</pre>	id *sendbuf, int sendcount,	37 38
		<pre>pe, void *recvbuf, const int recvcounts[],</pre>	39
		MPI_Datatype recvtype, MPI_Comm comm,	40
	MPI_Info info, MPI_	Request *request)	41
int N	-	void *sendbuf, MPI_Count sendcount,	42
	MPI_Datatype sendty	-	43
		vcounts[], const MPI_Aint displs[], pe, MPI_Comm comm, MPI_Info info,	44 45
	MPI_Datatype recvty MPI_Request *reques	-	45 46
			47
			48

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
3
                   displs, recvtype, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER, INTENT(IN) :: sendcount
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
14
                   displs, recvtype, comm, info, request, ierror) !(_c)
15
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
16
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
17
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
18
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
20
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Info), INTENT(IN) :: info
23
         TYPE(MPI_Request), INTENT(OUT) :: request
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     Fortran binding
27
     MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
28
                   DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
29
         <type> SENDBUF(*), RECVBUF(*)
30
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
31
                    INFO, REQUEST, IERROR
32
         Creates a persistent collective communication request for the allgathery operation.
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

## 6.13.6 Persistent All-to-All Scatter/Gather

0.15.0	Fersistent All-to-All Sc		2
			3
MPI_		endcount, sendtype, recvbuf, recvcount, recvtype, comm,	4 5
	info, request)		6
IN	sendbuf	starting address of send buffer (choice)	7
IN	sendcount	number of elements sent to each process	8
		(non-negative integer)	9
IN	sendtype	datatype of send buffer elements (handle)	10 11
OU	recvbuf	address of receive buffer (choice)	11
IN	recvcount	number of elements received from any process	13
		(non-negative integer)	14
IN	recvtype	datatype of receive buffer elements (handle)	15
IN	comm	communicator (handle)	16 17
IN	info	info argument (handle)	18
OU <sup>-</sup>		communication request (handle)	19
00	request	communication request (nanoic)	20
C bi	nding		21
	0	void *sendbuf, int sendcount,	22
		sendtype, void *recvbuf, int recvcount,	23 24
	MPI_Datatype n	recvtype, MPI_Comm comm, MPI_Info info,	24 25
	MPI_Request *1	request)	26
<pre>int MPI_Alltoall_init_c(const void *sendbuf, MPI_Count sendcount,</pre>			27
MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,			28
			29
	MPI_Request *1	request)	30
Forti	an 2008 binding		31
	0	sendcount, sendtype, recvbuf, recvcount,	32 33
	recvtype, comm	n, info, request, ierror)	34
1	<pre>YPE(*), DIMENSION(),</pre>	INTENT(IN), ASYNCHRONOUS :: sendbuf	35
	NTEGER, INTENT(IN) ::	-	36
	• -	ENT(IN) :: sendtype, recvtype	37
		ASYNCHRONOUS :: recvbuf	38
	YPE(MPI_Comm), INTENT(		39
	YPE(MPI_Info), INTENT( YPE(MPI_Request), INTE		40
	ITEGER, OPTIONAL, INTE	-	41
			42
MPI_A		sendcount, sendtype, recvbuf, recvcount,	43
_	• =	n, info, request, ierror) !(_c)	44
		INTENT(IN), ASYNCHRONOUS :: sendbuf	45 46
		KIND), INTENT(IN) :: sendcount, recvcount	40 47
		<pre>ENT(IN) :: sendtype, recvtype ASYNCHRONOUS :: recvbuf</pre>	48
1		Normonood Ieerbat	

289

1 2 3 4	TYPE TYPE	(MPI_Comm), INTENT(IN) :: (MPI_Info), INTENT(IN) :: (MPI_Request), INTENT(OUT) GER, OPTIONAL, INTENT(OUT)	info ) :: request		
5 6 7 8 9 10 11	Fortran binding MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR</type>				
12 13 14	Creat	es a persistent collective comm	nunication request for the alltoall operation.		
15 16	MPI_ALLT	OALLV_INIT(sendbuf, sendcou recvtype, comm, info, req	ints, sdispls, sendtype, recvbuf, recvcounts, rdispls, juest)		
17	IN	sendbuf	starting address of send buffer (choice)		
18 19 20 21	IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank		
22 23 24	IN	sdispls	Integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j		
25	IN	sendtype	datatype of send buffer elements (handle)		
26	OUT	recvbuf	address of receive buffer (choice)		
27 28 29 30	IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank		
31 32 33	IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i		
34	IN	recvtype	datatype of receive buffer elements (handle)		
35 36	IN	comm	communicator (handle)		
37	IN	info	info argument (handle)		
38 39	OUT	request	communication request (handle)		
40 41 42 43 44 45 46 47	C binding int MPI_Alltoallv_init(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request)				
47	int MPI_A		id *sendbuf, const MPI_Count sendcounts[], ls[], MPI_Datatype sendtype,		

292	CH	HAPTER 6. COLLECTIVE COMMUNICATION		
MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)				
IN	sendbuf	starting address of send buffer (choice)		
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)		
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)		
IN	sendtypes	Array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)		
OUT	recvbuf	address of receive buffer (choice)		
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)		
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)		
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)		
IN	comm	communicator (handle)		
IN	info	info argument (handle)		
OUT	request	communication request (handle)		
<pre>C binding int MPI_Alltoallw_init(const void *sendbuf, const int sendcounts[],</pre>				
<pre>int MPI_Alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>				
<pre>Fortran 2008 binding MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,</pre>				
	MPI_ALLT IN IN IN IN OUT IN IN IN IN IN OUT C binding int MPI_A int MPI_A	<pre>MPI_ALLTOALLW_INIT(sendbuf, sendcourecvtypes, comm, info, re IN sendbuf IN sendcounts IN sdispls IN sdispls IN sendtypes OUT recvbuf IN recvcounts IN rdispls IN rdispls IN rdispls IN recvtypes IN comm IN info OUT request C binding int MPI_Alltoallw_init(const void</pre>		

```
INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                    1
                                                                                    2
               recvcounts(*), rdispls(*)
                                                                                    3
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                    4
               recvtypes(*)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    5
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    6
                                                                                    7
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    9
                                                                                    10
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                    11
              recvcounts, rdispls, recvtypes, comm, info, request, ierror)
                                                                                    12
              !(_c)
                                                                                    13
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    14
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                    15
               sendcounts(*), recvcounts(*)
                                                                                    16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                    17
              rdispls(*)
                                                                                    18
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                    19
               recvtypes(*)
                                                                                    20
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                    21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    22
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                    23
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    25
                                                                                    26
Fortran binding
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                    27
                                                                                    28
              RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
                                                                                    29
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    30
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
                                                                                    31
               RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
                                                                                    32
    Creates a persistent collective communication request for the alltoally operation.
                                                                                    33
                                                                                    34
                                                                                    35
                                                                                    36
                                                                                    37
                                                                                    38
                                                                                    39
                                                                                    40
                                                                                    41
                                                                                    42
                                                                                    43
                                                                                    44
                                                                                    45
```

```
1
     6.13.7 Persistent Reduce
\mathbf{2}
3
4
     MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, info, request)
5
       IN
                sendbuf
                                            address of send buffer (choice)
6
       OUT
7
                 recvbuf
                                            address of receive buffer (choice, significant only at
8
                                            root)
9
       IN
                count
                                            number of elements in send buffer (non-negative
10
                                            integer)
11
       IN
                datatype
                                            datatype of elements of send buffer (handle)
12
       IN
                                            reduce operation (handle)
13
                ор
14
       IN
                root
                                            rank of root process (integer)
15
       IN
                                            communicator (handle)
                comm
16
17
       IN
                info
                                            info argument (handle)
18
       OUT
                 request
                                            communication request (handle)
19
20
     C binding
21
     int MPI_Reduce_init(const void *sendbuf, void *recvbuf, int count,
22
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
23
                    MPI_Info info, MPI_Request *request)
24
25
     int MPI_Reduce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
26
                    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
                    MPI_Info info, MPI_Request *request)
27
28
     Fortran 2008 binding
29
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
30
                    request, ierror)
31
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
32
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
33
          INTEGER, INTENT(IN) :: count, root
34
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
          TYPE(MPI_Op), INTENT(IN) :: op
36
          TYPE(MPI_Comm), INTENT(IN) :: comm
37
          TYPE(MPI_Info), INTENT(IN) :: info
38
          TYPE(MPI_Request), INTENT(OUT) :: request
39
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
42
                    request, ierror) !(_c)
43
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
44
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
45
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
46
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
          TYPE(MPI_Op), INTENT(IN) :: op
48
          INTEGER, INTENT(IN) :: root
```

TYPE TYPE	(MPI_Comm), INTENT(IN) :: (MPI_Info), INTENT(IN) :: (MPI_Request), INTENT(OUT) GER, OPTIONAL, INTENT(OUT)	info ) :: request	1 2 3 4
<type< td=""><td>CE_INIT(SENDBUF, RECVBUF, REQUEST, IERROR) e&gt; SENDBUF(*), RECVBUF(*)</td><td>COUNT, DATATYPE, OP, ROOT, COMM, INFO, ROOT, COMM, INFO, REQUEST, IERROR</td><td>5 6 7 8 9</td></type<>	CE_INIT(SENDBUF, RECVBUF, REQUEST, IERROR) e> SENDBUF(*), RECVBUF(*)	COUNT, DATATYPE, OP, ROOT, COMM, INFO, ROOT, COMM, INFO, REQUEST, IERROR	5 6 7 8 9
Creat	es a persistent collective comr	nunication request for the reduce operation.	11 12
6.13.8 P	ersistent All-Reduce		13 14 15
MPI_ALLF	REDUCE_INIT(sendbuf, recvbut	f, count, datatype, op, comm, info, request)	16 17
IN	sendbuf	starting address of send buffer (choice)	18
OUT	recvbuf	starting address of receive buffer (choice)	19
IN	count	number of elements in send buffer (non-negative integer)	20 21 22
IN	datatype	datatype of elements of send buffer (handle)	23
IN	ор	operation (handle)	24
IN	comm	communicator (handle)	25 26
IN	info	info argument (handle)	20
OUT	request	communication request (handle)	28
		- 、 /	29 30
C bindin	-	the second the second intervent	31
int MPI_		<pre>*sendbuf, void *recvbuf, int count, e, MPI_Op op, MPI_Comm comm,</pre>	32
	MPI_Info info, MPI_R		33
int MPT	Allreduce init c(const vo	id *sendbuf, void *recvbuf,	34 35
III0 III I_/		_Datatype datatype, MPI_Op op,	36
		nfo info, MPI_Request *request)	37
Fortran 3	2008 binding		38
	8	uf, count, datatype, op, comm, info,	39
	request, ierror)		40 41
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf			
	(*), DIMENSION(), ASYNC	HRONOUS :: recvbuf	43
	GER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN)	) :: datatype	44
	(MPI_Op), INTENT(IN) :: o		45
	(MPI_Comm), INTENT(IN) ::		46 47
TYPE	(MPI_Info), INTENT(IN) ::	info	47

12		(MPI_Request), INTENT(OU) GER, OPTIONAL, INTENT(OU)	<b>▲</b>
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	MPI_Allr TYPE TYPE INTE TYPE TYPE TYPE TYPE INTE	<pre>educe_init(sendbuf, recvi request, ierror) !( (*), DIMENSION(), INTEN (*), DIMENSION(), ASYNG GER(KIND=MPI_COUNT_KIND), (MPI_Datatype), INTENT(IN) (MPI_Op), INTENT(IN) :: ( (MPI_Comm), INTENT(IN) :: ( (MPI_Comm), INTENT(IN) :: ( (MPI_Info), INTENT(IN) :: ( (MPI_Request), INTENT(OUT))); ( GER, OPTIONAL, INTENT(OUT)); ()); ()); ()); ()); ()); ()); ());</pre>	<pre>puf, count, datatype, op, comm, info, _c) WT(IN), ASYNCHRONOUS :: sendbuf CHRONOUS :: recvbuf , INTENT(IN) :: count W) :: datatype op : comm : info F) :: request</pre>
18	<typ< td=""><td>e&gt; SENDBUF(*), RECVBUF(*)</td><td></td></typ<>	e> SENDBUF(*), RECVBUF(*)	
19 20	INTE	GER COUNT, DATATYPE, OP,	COMM, INFO, REQUEST, IERROR
20 21	Crea	tes a persistent collective com	munication request for the all reduce operation.
22 23 24 25 26 27		Persistent Reduce-Scatter wit PUCE_SCATTER_BLOCK_INIT info, request)	h Equal Blocks F(sendbuf, recvbuf, recvcount, datatype, op, comm,
28	IN	sendbuf	starting address of send buffer (choice)
29 30	OUT	recvbuf	starting address of receive buffer (choice)
31	IN	recvcount	element count per block (non-negative integer)
32 33 34	IN	datatype	datatype of elements of send and receive buffers (handle)
35	IN	ор	operation (handle)
36	IN	comm	communicator (handle)
37	IN	info	info argument (handle)
38 39	OUT	request	communication request (handle)
40 41 42 43 44	C bindir int MPI_	Reduce_scatter_block_init int recvcount, MPI_J	c(const void *sendbuf, void *recvbuf, Datatype datatype, MPI_Op op, Info info, MPI_Request *request)
45 46 47 48	int MPI_	MPI_Count recvcount	z_c(const void *sendbuf, void *recvbuf, , MPI_Datatype datatype, MPI_Op op, Info info, MPI_Request *request)

```
Fortran 2008 binding
                                                                                     1
                                                                                     2
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
                                                                                     3
              comm, info, request, ierror)
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                     4
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                     5
                                                                                     6
    INTEGER, INTENT(IN) :: recvcount
                                                                                     7
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                     8
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     9
                                                                                     10
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                     11
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     12
                                                                                     13
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
                                                                                     14
              comm, info, request, ierror) !(_c)
                                                                                     15
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                     16
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                     17
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                     18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                     19
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                     20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                     22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                     23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     ^{24}
                                                                                     25
Fortran binding
                                                                                     26
MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
                                                                                     27
              COMM, INFO, REQUEST, IERROR)
                                                                                     28
    <type> SENDBUF(*), RECVBUF(*)
                                                                                     29
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
                                                                                     30
    Creates a persistent collective communication request for the reduce-scatter with equal
                                                                                     ^{31}
blocks operation.
                                                                                     32
                                                                                     33
                                                                                     34
                                                                                     35
                                                                                     36
                                                                                     37
                                                                                     38
                                                                                     39
                                                                                     40
                                                                                     41
                                                                                     42
```

```
298
                                        CHAPTER 6. COLLECTIVE COMMUNICATION
1
     6.13.10 Persistent Reduce-Scatter
\mathbf{2}
3
4
     MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info,
5
                    request)
6
       IN
                sendbuf
                                            starting address of send buffer (choice)
7
       OUT
                recvbuf
8
                                            starting address of receive buffer (choice)
9
       IN
                recvcounts
                                            non-negative integer array specifying the number of
10
                                            elements in result distributed to each process. This
11
                                            array must be identical on all calling processes.
12
       IN
                datatype
                                            datatype of elements of input buffer (handle)
13
       IN
14
                op
                                            operation (handle)
15
       IN
                comm
                                            communicator (handle)
16
                info
       IN
                                            info argument (handle)
17
       OUT
18
                request
                                            communication request (handle)
19
20
     C binding
21
     int MPI_Reduce_scatter_init(const void *sendbuf, void *recvbuf,
22
                    const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
23
                    MPI_Comm comm, MPI_Info info, MPI_Request *request)
24
     int MPI_Reduce_scatter_init_c(const void *sendbuf, void *recvbuf,
25
                    const MPI_Count recvcounts[], MPI_Datatype datatype,
26
                    MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)
27
28
     Fortran 2008 binding
29
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
30
                    info, request, ierror)
^{31}
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
32
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
33
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Op), INTENT(IN) :: op
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Info), INTENT(IN) :: info
38
         TYPE(MPI_Request), INTENT(OUT) :: request
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
41
                    info, request, ierror) !(_c)
42
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
44
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         TYPE(MPI_Op), INTENT(IN) :: op
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

TYP	-	T(IN) :: info TENT(OUT) :: request TENT(OUT) :: ierror	1 2 3		
MPI_RED <ty INT</ty 	INFO, REQUES pe> SENDBUF(*), RE EGER RECVCOUNTS(*)		4 5 7 8 9		
	Persistent Inclusive		11 12 13 14		
MPI_SC/	AN_INIT(sendbuf, recv	buf, count, datatype, op, comm, info, request)	15 16		
IN	sendbuf	starting address of send buffer (choice)	17		
OUT	recvbuf	starting address of receive buffer (choice)	18		
IN	count	number of elements in input buffer (non-negative integer)	19 20 21		
IN	datatype	datatype of elements of input buffer (handle)	22		
IN	ор	operation (handle)	23		
IN	comm	communicator (handle)	24		
IN	info	info argument (handle)	25 26		
OUT	request	communication request (handle)	27		
			28 29		
C bindi	0		30		
int MPI		oid *sendbuf, void *recvbuf, int count, e datatype, MPI_Op op, MPI_Comm comm,	31		
	• -	o, MPI_Request *request)	32		
·+ MDT			33		
int MPI		<pre>void *sendbuf, void *recvbuf, MPI_Count count, e datatype, MPI_Op op, MPI_Comm comm,</pre>	34 35		
		o, MPI_Request *request)	36		
Fontnon	2008 binding		37		
	0	cvbuf, count, datatype, op, comm, info, request,	38		
111 1_000	ierror)	over, count, accuspe, op, comm, into, requeet,	39		
TYP	E(*), DIMENSION(	), INTENT(IN), ASYNCHRONOUS :: sendbuf	40 41		
	TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf				
INTEGER, INTENT(IN) :: count					
	TYPE(MPI_Datatype), INTENT(IN) :: datatype TYPE(MPI_Op), INTENT(IN) :: op				
	TYPE(MPI_Op), INTENT(IN) :: comm				
	TYPE(MPI Info), INTENT(IN) :: info				
TYP	E(MPI_Request), IN	TENT(OUT) :: request	47 48		

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
3
                     ierror) !(_c)
4
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
6
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
7
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
          TYPE(MPI_Op), INTENT(IN) :: op
9
          TYPE(MPI_Comm), INTENT(IN) :: comm
10
          TYPE(MPI_Info), INTENT(IN) :: info
11
          TYPE(MPI_Request), INTENT(OUT) :: request
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
16
                     IERROR)
17
          <type> SENDBUF(*), RECVBUF(*)
18
          INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
19
          Creates a persistent collective communication request for the inclusive scan operation.
20
21
     6.13.12 Persistent Exclusive Scan
22
23
^{24}
     MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)
25
26
       IN
                 sendbuf
                                             starting address of send buffer (choice)
27
       OUT
                 recvbuf
                                             starting address of receive buffer (choice)
28
29
       IN
                                             number of elements in input buffer (non-negative
                 count
30
                                             integer)
^{31}
       IN
                 datatype
                                             datatype of elements of input buffer (handle)
32
       IN
                 ор
                                             operation (handle)
33
34
       IN
                 comm
                                             intra-communicator (handle)
35
       IN
                 info
                                             info argument (handle)
36
       OUT
                 request
                                             communication request (handle)
37
38
39
     C binding
     int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,
40
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
41
                    MPI_Info info, MPI_Request *request)
42
43
     int MPI_Exscan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
44
                    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
45
                    MPI_Info info, MPI_Request *request)
46
47
48
```

Fortran 2008 binding 1 2 MPI\_Exscan\_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, ierror) 4 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 5 6 INTEGER, INTENT(IN) :: count TYPE(MPI\_Datatype), INTENT(IN) :: datatype 7 TYPE(MPI\_Op), INTENT(IN) :: op TYPE(MPI\_Comm), INTENT(IN) :: comm 9 10 TYPE(MPI\_Info), INTENT(IN) :: info TYPE(MPI\_Request), INTENT(OUT) :: request 11 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1213 MPI\_Exscan\_init(sendbuf, recvbuf, count, datatype, op, comm, info, request, 14 ierror) !(\_c) 15TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 16TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 17INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: count 18 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 19 TYPE(MPI\_Op), INTENT(IN) :: op 20TYPE(MPI\_Comm), INTENT(IN) :: comm 21TYPE(MPI\_Info), INTENT(IN) :: info 22 TYPE(MPI\_Request), INTENT(OUT) :: request 23INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2425Fortran binding 26MPI\_EXSCAN\_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, 27IERROR) 28 <type> SENDBUF(\*), RECVBUF(\*) 29 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 30 Creates a persistent collective communication request for the exclusive scan operation. 3132 6.14 Correctness 33 34 A correct, portable program must invoke collective communications so that deadlock will not 35occur, whether collective communications are synchronizing or not. The following examples 36 illustrate dangerous use of collective routines on intra-communicators. 37 38 **Example 6.25** The following is erroneous. 39 40 /\* ----- THIS EXAMPLE IS ERRONEOUS ----- \*/ 41 switch(rank) { 42case 0: 43 MPI\_Bcast(buf1, count, type, 0, comm); 44 MPI\_Bcast(buf2, count, type, 1, comm); 45break; 46case 1: 47MPI\_Bcast(buf2, count, type, 1, comm);

```
MPI_Bcast(buf1, count, type, 0, comm);
    break;
}
We assume that the group of comm is {0,1}. Two processes execute two broadcast operations
in reverse order. If the operation is synchronizing then a deadlock will occur.
Collective operations must be executed in the same order at all members of the communi-
cation group.
```

 $^{24}$ 

 $\mathbf{2}$ 

**Example 6.26** The following is erroneous.

```
/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm0);
        MPI_Bcast(buf2, count, type, 2, comm2);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 0, comm0);
        break;
    case 2:
        MPI_Bcast(buf1, count, type, 2, comm2);
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf1, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 1, comm1);
        MPI_Bcast(buf2, count, type, 1, comm1);
        break;
}
```

Assume that the group of comm0 is {0,1}, of comm1 is {1, 2} and of comm2 is {2,0}. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependencies occur. Nonblocking collective operations can alleviate this issue.

```
Example 6.27 The following is erroneous.
```

```
/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, 0, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
```

}

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero may block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

```
Example 6.28 An unsafe, nondeterministic program.
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break:
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 6.12. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

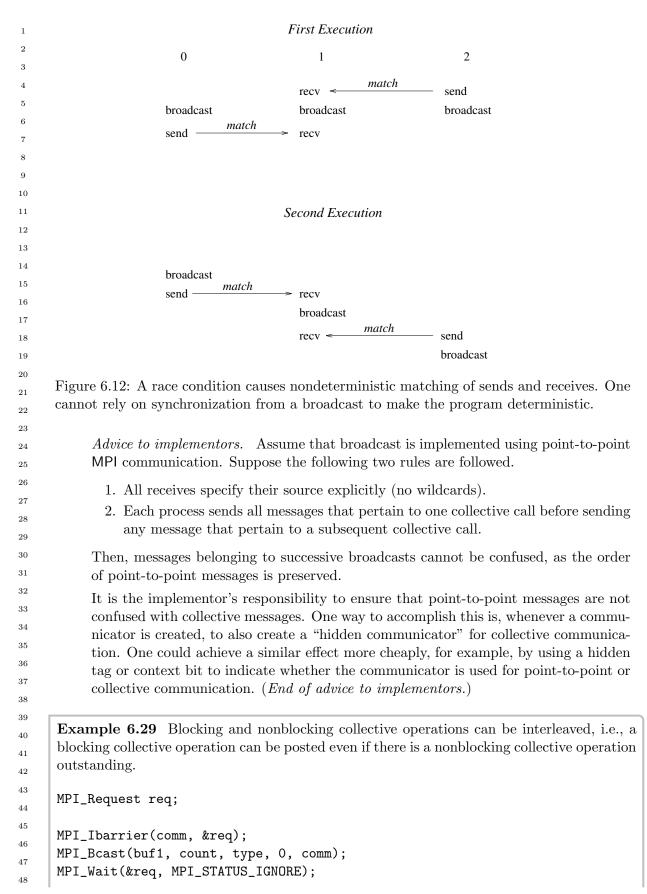
Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication initialization call at an MPI process. In these situations, it is the user's responsibility to ensure that the same communicator is not used concurrently by two different collective communication initialization calls at the same MPI process. Collective communication initialization calls include all calls for blocking collective operations, all initiation calls for nonblocking collective operations, and all initialization calls for persistent collective operations.

 $\mathbf{5}$ 

 $\overline{7}$ 

 $^{24}$ 

 $45 \\ 46$ 



Each process starts a nonblocking barrier operation, participates in a blocking broadcast and then waits until every other process started the barrier operation. This effectively turns the broadcast into a synchronizing broadcast with possible communication/communication overlap (MPI\_Bcast is allowed, but not required to synchronize).

**Example 6.30** The starting order of collective operations on a particular communicator defines their matching. The following example shows an erroneous matching of different collective operations on the same communicator.

```
THIS EXAMPLE IS ERRONEOUS ----- */
/* -----
MPI_Request req;
switch(rank) {
   case 0:
       /* erroneous matching */
       MPI_Ibarrier(comm, &req);
       MPI_Bcast(buf1, count, type, 0, comm);
       MPI_Wait(&req, MPI_STATUS_IGNORE);
       break;
    case 1:
       /* erroneous matching */
       MPI_Bcast(buf1, count, type, 0, comm);
       MPI_Ibarrier(comm, &req);
       MPI_Wait(&req, MPI_STATUS_IGNORE);
       break;
}
```

This ordering would match MPI\_Ibarrier on rank 0 with MPI\_Bcast on rank 1 which is erroneous and the program behavior is undefined. However, if such an order is required, the user must create different duplicate communicators and perform the operations on them. If started with two processes, the following program would be correct:

```
MPI_Request req;
MPI_Comm dupcomm;
MPI_Comm_dup(comm, &dupcomm);
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, dupcomm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 0, dupcomm);
        MPI_Ibarrier(comm, &req);
        MPI_Uait(&req, MPI_STATUS_IGNORE);
        break;
}
```

Advice to users. The use of different communicators offers some flexibility regarding

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26 27

28

29

30

31 32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

 $\mathbf{2}$ 

 the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (*End of advice to users.*)

# **Example 6.31** Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

```
MPI_Request req;
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        MPI_Send(buf, count, dtype, 1, tag, comm);
        break;
        case 1:
        MPI_Ibarrier(comm, &req);
        MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

The MPI library must *progress* the barrier in the MPI\_Recv call. Thus, the MPI\_Wait call in rank 0 will eventually complete, which enables the matching MPI\_Send so all calls eventually return.

**Example 6.32** Blocking and nonblocking collective operations do not match. The following example is erroneous.

```
/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
MPI_Request req;
switch(rank) {
   case 0:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
   case 1:
        /* erroneous false matching of Alltoall and Ialltoall */
        MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
        break;
}
```

**Example 6.33** Collective and point-to-point requests can be mixed in functions that enable multiple completions. If started with two processes, the following program is valid.

```
MPI_Request reqs[2];
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &reqs[0]);
        MPI_Send(buf, count, dtype, 1, tag, comm);
        MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
        break;
    case 1:
        MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
        MPI_Ibarrier(comm, &reqs[1]);
        MPI_Ibarrier(comm, &reqs[1]);
        MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
        break;
}
```

The MPI\_Waitall call returns only after the barrier and the receive completed.

**Example 6.34** Multiple nonblocking collective operations can be outstanding on a single communicator and match in order.

MPI\_Request reqs[3];

```
compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
compute(buf3);
MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
```

Advice to users. Pipelining and double-buffering techniques can efficiently be used to overlap computation and communication. However, having too many outstanding requests might have a negative impact on performance. (*End of advice to users.*)

Advice to implementors. The use of pipelining may generate many outstanding requests. A high-quality hardware-supported implementation with limited resources should be able to fall back to a software implementation if its resources are exhausted. In this way, the implementation could limit the number of outstanding requests only by the available memory. (End of advice to implementors.)

**Example 6.35** Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 6.13). The following example is started with three processes and three communicators. The first communicator comm1 includes ranks 0 and 1, comm2 includes ranks 1 and 2, and comm3 spans ranks 0

 $\mathbf{2}$ 

3 4

5 6

7

8

9

10

11

12

13

14

15

16

17

18

19 20 21

22

23 24

25 26

27

28

29

30

31

32

33 34

35

36

37

38 39

40 41

42

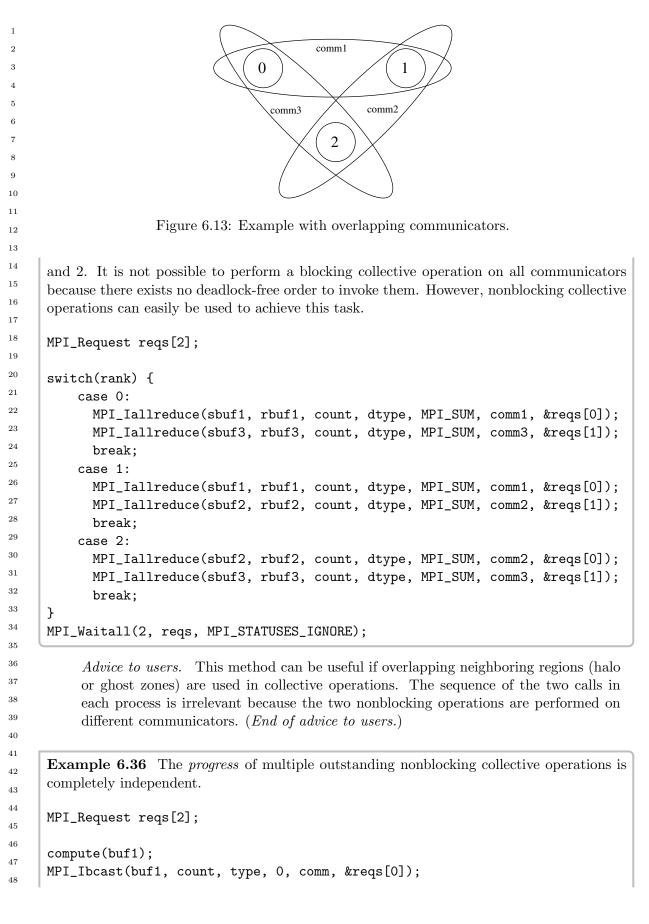
43 44

45

46

47

#### CHAPTER 6. COLLECTIVE COMMUNICATION



compute(buf2); MPI\_Ibcast(buf2, count, type, 0, comm, &reqs[1]); MPI\_Wait(&reqs[1], MPI\_STATUS\_IGNORE); /\* nothing is known about the status of the first bcast here \*/ MPI\_Wait(&reqs[0], MPI\_STATUS\_IGNORE);

Finishing the second MPI\_IBCAST is completely independent of the first one. This means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via reqs[1].

# Chapter 7

# Groups, Contexts, Communicators, and Caching

 $^{24}$ 

### 7.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a "higher level" of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [4] and [62] for further information on writing libraries in MPI, using the features described in this chapter.

#### 7.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.

### 7.1.2 MPI's Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- Groups of processes,
- Virtual topologies,
- Attribute caching,
- Communicators.

<sup>13</sup> <sup>14</sup> **Communicators** (see [22, 60, 64]) encapsulate all of these ideas in order to provide the <sup>15</sup> appropriate scope for all communication operations in MPI. Communicators are divided <sup>16</sup> into two kinds: intra-communicators for operations within a single group of processes and <sup>17</sup> inter-communicators for operations between two groups of processes.

<sup>19</sup> Caching. Communicators (see below) provide a "caching" mechanism that allows one to <sup>20</sup> associate new attributes with communicators, on par with MPI built-in features. This can <sup>21</sup> be used by advanced users to adorn communicators further, and by MPI to implement <sup>22</sup> some communicator functions. For example, the virtual-topology functions described in <sup>23</sup> Chapter 8 are likely to be supported this way.

24

1

2

3

4 5

6

7 8

9 10

11

12

18

Groups. Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

31

36

37

38

39

40

41

42

43

44

Intra-Communicators. The most commonly used means for message-passing in MPI is via
 intra-communicators. Intra-communicators contain an instance of a group, contexts of
 communication for both point-to-point and collective communication, and the ability to
 include virtual topology and other attributes. These features work as follows:

• **Contexts** provide the ability to have separate safe "universes" of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.

- 45 46
- **Groups** define the participants in the communication (see above) of a communicator.
- 47 48

- A virtual topology defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 8 to provide this feature. Intra-communicators as described in this chapter do not have topologies.
- Attributes define the local information that the user or library has added to a communicator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. When using the World Model (Section 11.2), this practice can be followed in MPI by using the predefined communicator MPI\_COMM\_WORLD. (End of advice to users.)

Inter-Communicators. The discussion has dealt so far with intra-communication: communication within a group. MPI also supports **inter-communication**: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a 20client-server computing paradigm, where either client or server are parallel. The support 21of inter-communication also provides a mechanism for the extension of MPI to a dynamic 22model where not all processes are preallocated at initialization time. In such a situation, 23it becomes necessary to support communication across "universes." Inter-communication is supported by objects called **inter-communicators**. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows: 27

- Contexts provide the ability to have a separate safe "universe" of message-passing between the two groups. A send operation in the local group is always matched by a receive operation in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code.
- A local and remote group specify the recipients and destinations for an inter-communicator.
- Virtual topology is undefined for an inter-communicator.
- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They 44are used for point-to-point and collective communication in a related manner to intra-4546communicators. Users who do not need inter-communication in their applications can safely ignore this extension. Users who require inter-communication between overlapping groups 4748 must layer this capability on top of MPI.

1  $\mathbf{2}$ 

3

4

5

6  $\overline{7}$ 

8

9

10

11

12

13

1415

16

17

18

19

 $^{24}$ 

25

26

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38 39

40 41

42

#### 7.2 **Basic Concepts**

In this section, we turn to a more formal definition of the concepts introduced above.

#### 7.2.1 Groups

6 A group is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects. Each process in a group is associated with an integer **rank**. Ranks are contiguous and start from zero. Groups are represented by opaque group objects, and hence cannot be directly transferred from one process to another. A group is 10 used within a communicator to describe the participants in a communication "universe" 11 and to rank such participants (thus giving them unique names within that "universe" of 12communication). 13

There is a special pre-defined group: MPI\_GROUP\_EMPTY, which is a group with no 14members. The predefined constant MPI\_GROUP\_NULL is the value used for invalid group 15handles. 16

MPI\_GROUP\_EMPTY, which is a valid handle to an empty group, Advice to users. should not be confused with MPI\_GROUP\_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (End of advice to users.)

22Advice to implementors. A group may be represented by a virtual-to-real process-23address-translation table. Each communicator object (see below) would have a pointer 24to such a table.

Simple implementations of MPI will enumerate groups, such as in a table. However, more advanced data structures make sense in order to improve scalability and memory usage with large numbers of processes. Such implementations are possible with MPI. (End of advice to implementors.)

#### 7.2.2 Contexts

32 A context is a property of communicators (defined next) that allows partitioning of the 33 communication space. A message sent in one context cannot be received in another context. 34Furthermore, where permitted, collective operations are independent of pending point-topoint operations. Contexts are not explicit MPI objects; they appear only as part of the 35 36 realization of communicators (below).

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don't interfere.

44A possible implementation for a context is as a supplemental tag attached to messages 45on send and matched on receive. Each intra-communicator stores the value of its two 46tags (one for point-to-point and one for collective communication). Communicator-47 generating functions use a collective communication to agree on a new group-wide 48 unique context.

1  $\mathbf{2}$ 3

45

7

8

9

17

18

19

20

21

25

26

27

28

29 30

 $^{31}$ 

37

38

39

40

41

42

Analogously, in inter-communication, two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (*End of advice to implementors.*)

#### 7.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 8), communicators may also "cache" additional information (see Section 7.7). MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message are identified by process ranks within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the "spatial" scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque **intra-communicator objects**, and hence cannot be directly transferred from one process to another.

#### 7.2.4 Predefined Intra-Communicators

When using the World Model (Section 11.2) for MPI initialization, an initial intra-communicator MPI\_COMM\_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI\_INIT or MPI\_INIT\_THREAD has been called. In addition, the communicator MPI\_COMM\_SELF is provided, which includes only the process itself. When using the Sessions Model (Section 11.3) for initialization of MPI resources, MPI\_COMM\_WORLD and MPI\_COMM\_SELF are not valid for use as a communicator. See the discussion concerning use of MPI named constants in 2.5.4 for valid uses of MPI\_COMM\_WORLD and MPI\_COMM\_SELF prior to initialization of MPI. See also the discussion concerning interoperability of the World Model and Sessions Model in Section 11.1.

The predefined constant MPI\_COMM\_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the computation are available after MPI is initialized. For this case, MPI\_COMM\_WORLD is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI\_COMM\_WORLD is a communicator incorporating all processes with which the joining process can immediately communicate. Therefore, MPI\_COMM\_WORLD may simultaneously represent disjoint groups in different processes.

All MPI implementations are required to provide the MPI\_COMM\_WORLD communicator. It cannot be deallocated during the life of a process. The group corresponding to this communicator does not appear as a pre-defined constant, but it may be accessed using 48

1

 $\mathbf{2}$ 

3

4

5

6 7

8 9

10

11 12

13

14 15

16

17

18

19

20

21

22 23 24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

MPI\_COMM\_GROUP (see below). MPI does not specify the correspondence between the
 process rank in MPI\_COMM\_WORLD and its (machine-dependent) absolute address. Neither
 does MPI specify the function of the host process, if any. Other implementation-dependent,
 predefined communicators may also be provided.

5 6 7

8

9

10 11

### 7.3 Group Management

This section describes the manipulation of process groups in MPI. These operations are local and their execution does not require interprocess communication.

```
7.3.1 Group Accessors
12
13
14
     MPI_GROUP_SIZE(group, size)
15
16
       IN
                                             group (handle)
                 group
17
       OUT
                 size
                                             number of processes in the group (integer)
18
19
     C binding
20
     int MPI_Group_size(MPI_Group group, int *size)
21
22
     Fortran 2008 binding
23
     MPI_Group_size(group, size, ierror)
^{24}
          TYPE(MPI_Group), INTENT(IN) :: group
25
          INTEGER, INTENT(OUT) :: size
26
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     Fortran binding
28
     MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
29
          INTEGER GROUP, SIZE, IERROR
30
^{31}
32
     MPI_GROUP_RANK(group, rank)
33
34
       IN
                                             group (handle)
                 group
35
       OUT
                 rank
                                             rank of the calling process in group, or
36
                                             MPI_UNDEFINED if the process is not a member
37
                                             (integer)
38
39
     C binding
40
     int MPI_Group_rank(MPI_Group group, int *rank)
41
42
     Fortran 2008 binding
43
     MPI_Group_rank(group, rank, ierror)
44
          TYPE(MPI_Group), INTENT(IN) :: group
45
          INTEGER, INTENT(OUT) :: rank
46
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

	binding P_RANK(GROUP, R. GER GROUP, RANK		1 2 3 4 5	
MPI_GRC	UP_TRANSLATE_	_RANKS(group1, n, ranks1, group2, ranks2)	6	
IN	group1	group1 (handle)	7 8	
IN	n	number of ranks in ranks1 and ranks2 arrays (integer)	9	
IN	ranks1	array of zero or more valid ranks in group1	10	
IN	group2	group2 (handle)	11	
OUT	ranks2	array of corresponding ranks in group2,	12 13	
001	141152	MPI_UNDEFINED when no correspondence exists.	13	
			15	
C bindir	ıg		16	
int MPI_	-	_ranks(MPI_Group group1, int n, const int ranks1[],	17 18	
	MP1_Group	group2, int ranks2[])	19	
	2008 binding		20	
		ks(group1, n, ranks1, group2, ranks2, ierror)	21	
	-	<pre>IENT(IN) :: group1, group2     :: n, ranks1(n)</pre>	22	
	GER, INTENT(IN) GER, INTENT(OUT)		23	
		INTENT(OUT) :: ierror	24	
			25 26	
Fortran	-	KS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)	27	
		RANKS1(*), GROUP2, RANKS2(*), IERROR	28	
			29	
	-	ant for determining the relative numbering of the same processes	30	
	· ·	instance, if one knows the ranks of certain processes in the group might want to know their ranks in a subset of that group.	31	
	,	lid rank for input to MPI_GROUP_TRANSLATE_RANKS, which	32	
		the translated rank.	33 34	
			35	
		oup1, group2, result)	36	
	ζ-	,	37	
IN	group1	first group (handle)	38	
IN	group2	second group (handle)	39 40	
OUT	OUT result (integer)			
C binding			43	
<pre>int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)</pre>			44	
Fortran 2008 binding			45	
MPI_Group_compare(group1, group2, result, ierror)				
TYPE(MPI_Group), INTENT(IN) :: group1, group2				
INTEGER, INTENT(OUT) :: result				

INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI\_GROUP\_COMPARE(GROUP1, GROUP2, RESULT, IERROR)

INTEGER GROUP1, GROUP2, RESULT, IERROR

MPI\_IDENT results if the group members and group order are exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI\_SIMILAR results if the group members are the same but the order is different. MPI\_UNEQUAL results otherwise.
7.3.2 Group Constructors

MPI provides two approaches to constructing groups. In the first approach MPI procedures

MPI provides two approaches to constructing groups. In the first approach, MPI procedures are provided to subset and superset existing groups. These constructors construct new groups from existing groups. In the second approach, a group is created using a session handle and associated process set. This second approach is available when using the Sessions Model. With both approaches, these are local operations, and distinct groups may be defined on different processes; a process may also define a group that does not include itself. Consistent definitions are required when groups are used as arguments in communicator creation functions. When using the World Model (Section 11.2) for MPI initialization, the base group, upon which all other groups are defined, is the group associated with the initial communicator MPI\_COMM\_WORLD (accessible through the function MPI\_COMM\_GROUP). 

- Rationale. In what follows, there is no group duplication function analogous to MPI\_COMM\_DUP, defined later in this chapter. There is no need for a group duplicator. A group, once created, can have several references to it by making copies of the handle. The following constructors address the need for subsets and supersets of existing groups. (End of rationale.)
- Advice to implementors. Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (End of advice to implementors.)

<sup>36</sup> MPI\_COMM\_GROUP(comm, group)

 $\overline{7}$ 

```
37
       IN
                                            communicator (handle)
                comm
38
       OUT
                group
                                            group corresponding to comm (handle)
39
40
41
     C binding
42
     int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
43
     Fortran 2008 binding
44
     MPI_Comm_group(comm, group, ierror)
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         TYPE(MPI_Group), INTENT(OUT) :: group
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

Fortran binding				
MPI_COMM_GROUP(COMM, GROUP, IERROR)				
INTEGER COMM, GROUP, IERROR				
MPI_	_COMM_GROUP returns in g	group a handle to the group of comm.	4	
			6	
		<b>、</b>	7	
MPI_GRC	OUP_UNION(group1, group2,	newgroup)	8	
IN	group1	first group (handle)	9	
IN	group2	second group (handle)	10	
OUT	newgroup	union group (handle)	11	
001	newgroup	union group (nondic)	12	
C bindiı	າອ		13	
	_Group_union(MPI_Group gi	roup1, MPI_Group group2,	14	
	MPI_Group *newgrou		15	
<b>D</b>		•	16 17	
	2008 binding	·····	17	
	<pre>up_union(group1, group2, C(MPI_Group), INTENT(IN)</pre>		19	
	C(MPI_Group), INTENT(IN)		20	
	GER, OPTIONAL, INTENT(OU	0 1	21	
			22	
Fortran	6		23	
	JP_UNION(GROUP1, GROUP2, GER GROUP1, GROUP2, NEW	-	24	
	GER GROUFI, GROUFZ, NEW	RUOP, IERROR	25	
			26	
MPL GRO	OUP_INTERSECTION(group1	group? newgroup)	27	
		,	28 29	
IN	group1	first group (handle)	30	
IN	group2	second group (handle)	31	
OUT	newgroup	intersection group (handle)	32	
			33	
C bindi	ng		34	
int MPI_	-	Group group1, MPI_Group group2,	35	
	MPI_Group *newgrou	p)	36	
Fortran	2008 binding		37	
	e	group2, newgroup, ierror)	38 39	
	C(MPI_Group), INTENT(IN)		40	
	C(MPI_Group), INTENT(OUT)		41	
INTE	GER, OPTIONAL, INTENT(OU	JT) :: ierror	42	
Fortran binding			43	
			44	
			45	
			46	
			47	
	4			

```
1
      MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
\mathbf{2}
       IN
                  group1
                                               first group (handle)
3
       IN
                 group2
                                               second group (handle)
4
5
       OUT
                  newgroup
                                               difference group (handle)
6
\overline{7}
      C binding
8
      int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
9
                     MPI_Group *newgroup)
10
     Fortran 2008 binding
11
     MPI_Group_difference(group1, group2, newgroup, ierror)
12
          TYPE(MPI_Group), INTENT(IN) :: group1, group2
13
          TYPE(MPI_Group), INTENT(OUT) :: newgroup
14
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
      Fortran binding
17
     MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
18
          INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
19
      The set-like operations are defined as follows:
20
21
      union All elements of the first group (group1), followed by all elements of second group
22
           (group2) not in the first group.
23
24
     intersect All elements of the first group that are also in the second group, ordered as in
25
           the first group.
26
      difference All elements of the first group that are not in the second group, ordered as in
27
           the first group.
28
29
      Note that for these operations the order of processes in the output group is determined
30
      primarily by order in the first group (if possible) and then, if necessary, by order in the
^{31}
      second group. Neither union nor intersection are commutative, but both are associative.
32
      The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
33
34
35
     MPI_GROUP_INCL(group, n, ranks, newgroup)
36
       IN
                                               group (handle)
                 group
37
       IN
                                               number of elements in array ranks (and size of
38
                  n
39
                                               newgroup) (integer)
40
       IN
                  ranks
                                               ranks of processes in group to appear in newgroup
41
                                               (array of integers)
42
       OUT
                                               new group derived from above, in the order defined
                  newgroup
43
                                               by ranks (handle)
44
45
     C binding
46
      int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
47
                     MPI_Group *newgroup)
48
```

Fortran	2008 binding		1
	p_incl(group, n, ranks, n	ewgroup, ierror)	2
TYPE	(MPI_Group), INTENT(IN) :	: group	3
INTE	GER, INTENT(IN) :: n, ran	ks(n)	4
TYPE	(MPI_Group), INTENT(OUT)	:: newgroup	5
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	6
Fortran	hinding		7
	P_INCL(GROUP, N, RANKS, N	EWGROUP, TERROR)	8
	GER GROUP, N, RANKS(*), N	-	9
			10 11
		reates a group newgroup that consists of the n pro-	11
-		anks[n-1]; the process with rank i in newgroup is the	12
-	•••	ch of the n elements of ranks must be a valid rank	14
• ·		inct, or else the program is erroneous. If $n = 0$ , This function can, for instance, be used to reorder	15
-	nts of a group. See also MPI_G		16
the eleme.	$115$ of a group. See also with $1_{-1}$	GROOT_COMPARE.	17
			18
MPI_GRO	UP_EXCL(group, n, ranks, nev	vgroup)	19
IN	group	group (handle)	20
IN	n	number of elements in array ranks (integer)	21
IN	ranks	array of integer ranks of processes in group not to	22 23
IIN	Tanks	appear in newgroup	23 24
			25
OUT	newgroup	new group derived from above, preserving the order defined by group (handle)	26
		defined by group (fiandle)	27
Chindin			28
C bindin	0	p, int n, const int ranks[],	29
IIIC MFI_	MPI_Group *newgroup)	-	30
			31
	2008 binding		32
	p_excl(group, n, ranks, n		33
	(MPI_Group), INTENT(IN) :	• ·	34
	GER, INTENT(IN) :: n, ran		35
	(MPI_Group), INTENT(OUT)		36
INTE	GER, OPTIONAL, INTENT(OUT	) :: lerror	37
Fortran	binding		38
MPI_GROU	P_EXCL(GROUP, N, RANKS, N	EWGROUP, IERROR)	$\frac{39}{40}$
INTE	GER GROUP, N, RANKS(*), N	EWGROUP, IERROR	40 41
The f	unction MPL GROUP FXCL of	reates a group of processes <b>newgroup</b> that is obtained	42
			43
by deleting from group those processes with ranks $ranks[0], \ldots, ranks[n-1]$ . The ordering of <sup>43</sup>			

The function MPI\_GROUP\_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0],..., ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct; otherwise, the program is erroneous. If n = 0, then newgroup is identical to group. 42 43 44 45 46

322 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

```
1
      MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
2
        IN
                                                  group (handle)
                   group
3
        IN
                   n
                                                  number of triplets in array ranges (integer)
4
5
        IN
                                                  a one-dimensional array of integer triplets, of the
                   ranges
6
                                                  form (first rank, last rank, stride) indicating ranks in
\overline{7}
                                                  group of processes to be included in newgroup
8
        OUT
                                                  new group derived from above, in the order defined
                   newgroup
9
                                                  by ranges (handle)
10
11
      C binding
12
      int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
13
                       MPI_Group *newgroup)
14
15
      Fortran 2008 binding
16
      MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
17
           TYPE(MPI_Group), INTENT(IN) :: group
18
           INTEGER, INTENT(IN) :: n, ranges(3, n)
19
           TYPE(MPI_Group), INTENT(OUT) :: newgroup
20
           INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
      Fortran binding
22
      MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
23
           INTEGER GROUP, N, RANGES(3, *), NEWGROUP, IERROR
^{24}
25
      If ranges consists of the triplets
26
            (first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)
27
28
      then newgroup consists of the sequence of processes in group with ranks
29
            first_1, first_1 + stride_1, \dots, first_1 + \left| \frac{last_1 - first_1}{stride_1} \right| stride_1, \dots,
30
^{31}
32
            first_n, first_n + stride_n, \dots, first_n + \left| \frac{last_n - first_n}{stride_n} \right| stride_n.
33
34
35
           Each computed rank must be a valid rank in group and all computed ranks must be
36
      distinct, or else the program is erroneous. Note that we may have first_i > last_i, and stride_i
37
      may be negative, but cannot be zero.
38
           The functionality of this routine is specified to be equivalent to expanding the array
39
      of ranges to an array of the included ranks and passing the resulting array of ranks and
40
      other arguments to MPI_GROUP_INCL. A call to MPI_GROUP_INCL is equivalent to a call
41
      to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i,i,1) in the
42
      argument ranges.
```

43 44

45

46

47

MPI_GROU	JP_RANGE_EXCL(group, n, ra	inges, newgroup)	1
IN	group	group (handle)	2 3
IN	n	number of triplets in array ranges (integer)	4
IN	ranges	a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in	5 6
		group of processes to be excluded from the output group newgroup (array of integers)	7 8
OUT	newgroup	new group derived from above, preserving the order	9 10
		in group (handle)	11
			12

# MPL GROUP RANGE EXCL(group n ranges newgroup)

#### C binding

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)
```

# Fortran 2008 binding

0		
MPI_Group_range_excl(gro	<pre>up, n, ranges, newgroup,</pre>	ierror)
TYPE(MPI_Group), INT	ENT(IN) :: group	
INTEGER, INTENT(IN)	:: n, ranges(3, n)	
TYPE(MPI_Group), INT	ENT(OUT) :: newgroup	
INTEGER, OPTIONAL, I	NTENT(OUT) :: ierror	
TYPE(MPI_Group), INT	ENT(OUT) :: newgroup	

# Fortran binding

MPI_	_GROUP_	RANGE_	EXCL	(GROUP,	N,	RANG	ΕS,	NEWGRO	)UP,	IERROR
	INTEGE	R GROU	JP, N	, RANGE	S(3	, *),	NE	WGROUP	, IEH	RROR

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI\_GROUP\_EXCL. A call to MPI\_GROUP\_EXCL is equivalent to a call to MPI\_GROUP\_RANGE\_EXCL with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

The range operations do not explicitly enumerate ranks, and Advice to users. therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenenever possible, as high-quality implementations will take advantage of this fact. (End of advice to users.)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (End of advice to implementors.)

13

14

1516

17

23

 $^{24}$ 2526

27

28

29

30

 $^{31}$ 

32

33 34

3536

37

38

39

40

```
324 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING
```

1 MPI\_GROUP\_FROM\_SESSION\_PSET(session, pset\_name, newgroup)  $\mathbf{2}$ IN session session (handle) 3 IN pset\_name name of process set to use to create the new group 4 (string) 56 OUT new group derived from supplied session and process newgroup 7 set (handle) 8 9 C binding 10 int MPI\_Group\_from\_session\_pset(MPI\_Session session, const char \*pset\_name, 11 MPI\_Group \*newgroup) 12Fortran 2008 binding 13 MPI\_Group\_from\_session\_pset(session, pset\_name, newgroup, ierror) 14TYPE(MPI\_Session), INTENT(IN) :: session 15CHARACTER(LEN=\*), INTENT(IN) :: pset\_name 16TYPE(MPI\_Group), INTENT(OUT) :: newgroup 17INTEGER, OPTIONAL, INTENT(OUT) :: ierror 18 19 Fortran binding 20MPI\_GROUP\_FROM\_SESSION\_PSET(SESSION, PSET\_NAME, NEWGROUP, IERROR) 21INTEGER SESSION, NEWGROUP, IERROR 22CHARACTER\*(\*) PSET\_NAME 23The function MPI\_GROUP\_FROM\_SESSION\_PSET creates a group newgroup using the  $^{24}$ provided session handle and process set. The process set name must be one returned from 25an invocation of MPI\_SESSION\_GET\_NTH\_PSET using the supplied session handle. If the 26pset\_name does not exist, MPI\_GROUP\_NULL will be returned in the newgroup argument. 27As with other group constructors, MPI\_GROUP\_FROM\_SESSION\_PSET is a local function. 28See Section 11.3 for more information on sessions and process sets. 29 30 317.3.3 Group Destructors 32 33 34MPI\_GROUP\_FREE(group) 35 INOUT group group (handle) 36 37 C binding 38 int MPI\_Group\_free(MPI\_Group \*group) 39 40Fortran 2008 binding 41 MPI\_Group\_free(group, ierror) 42TYPE(MPI\_Group), INTENT(INOUT) :: group 43 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 44Fortran binding 45MPI\_GROUP\_FREE(GROUP, IERROR) 46INTEGER GROUP, IERROR 4748

This operation marks a group object for deallocation. The handle group is set to MPI\_GROUP\_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI\_COMM\_GROUP, MPI\_COMM\_CREATE, MPI\_COMM\_DUP, MPI\_COMM\_IDUP, MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_IDUP\_WITH\_INFO, MPI\_COMM\_SPLIT, MPI\_COMM\_SPLIT\_TYPE, MPI\_COMM\_CREATE\_GROUP, MPI\_COMM\_CREATE\_FROM\_GROUP, MPI\_INTERCOMM\_CREATE, and MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS, and decremented for each call to MPI\_GROUP\_FREE or MPI\_COMM\_FREE; the group object is ultimately deallocated when the reference count drops to zero. (*End of advice to implementors.*)

# 7.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (*End of advice to implementors.*)

# 7.4.1 Communicator Accessors

The following are all local operations.

#### MPI\_COMM\_SIZE(comm, size)

IN	comm	communicator (handle)
OUT	size	number of processes in the group of comm (integer)

#### C binding

int MPI\_Comm\_size(MPI\_Comm comm, int \*size)

#### Fortran 2008 binding

MPI\_Comm\_size(comm, size, ierror)
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(OUT) :: size
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_COMM\_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR  $^{24}$ 

	326	C	CHAPTER 7.	GROUPS, CO	NTEXTS, COMMU	UNICATORS, A	ND CACHING
1 2 3 4 5		then	COMM_GRO	UP (see above), emporary group	alent to accessing to computing the size via MPI_GROUP_F ut was introduced.	using MPI_GROREE. However,	OUP_SIZE, and this function is
6 7 8 9 10		comn availa descr	able unless th ibed in Chap	or MPI_COMM_V ne number of pr oter 11; note that	indicates the num /ORLD, it indicates occesses has been c at the number of pr an MPI program.	the total numb hanged by using	er of processes g the functions
11 12 13 14 15 16		availa indica	able for a spe ates the rank	ecific library or of the process t	ext call to determi program. The follo hat calls it in the r MM_SIZE.( <i>End of c</i>	wing call, $MPI_{ange}$ from $0, \ldots$	COMM_RANK, size-1, where
17 18	MPI	СОМІ	M_RANK(con	nm rank)			
19	IN		comm		communicator (han	dle)	
20 21	OL	JT	rank		rank of the calling p	,	f comm (integer)
22 23 24		inding MPI_C	-	I_Comm comm, :	int *rank)		
25 26 27 28 29	MPI_	Comm_ TYPE( INTEG	MPI_Comm), ER, INTENT(	g rank, ierror) INTENT(IN) :: OUT) :: rank L, INTENT(OUT)			
30 31 32 33 34	MPI_	COMM_	inding RANK(COMM, ER COMM, RA	RANK, IERROR) NK, IERROR			
35 36 37 38 39		and t	COMM_GRO	UP (see above) ne temporary gro	ralent to accessing t computing the ra pup via MPI_GROUF ctcut was introduced	nk using MPI_0 P_FREE. Howeve	GROUP_RANK, er, this function
40 41				0	es the rank of the price above, in conjur	-	
42 43 44 45 46 47		Many mode role, two p	y programs v el, where one and the othe preceding call	vill be written process (such er processes will	with the supervisor as the rank-zero pr serve as compute : determining the role	r/executor or n cocess) will play nodes. In this	nanager/worker v a supervisory framework, the
48							

MPI_COM	IM_COMPARE(comm1, comm2	2. result)			
IN	comm1	first communicator (handle)			
IN	comm2	second communicator (handle)			
OUT	result	result (integer)			
C bindin int MPI_(	-	m1, MPI_Comm comm2, int *result)			
MPI_Comm TYPE INTE(	<pre>Fortran 2008 binding MPI_Comm_compare(comm1, comm2, result, ierror)     TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2     INTEGER, INTENT(OUT) :: result     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
MPI_COMM	Fortran binding MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR) INTEGER COMM1, COMM2, RESULT, IERROR				
MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.					
7.4.2 Co	mmunicator Constructors				

The following are collective functions that are invoked by all processes in the group or groups associated with comm, with the exception of MPI\_COMM\_CREATE\_GROUP, MPI\_COMM\_CREATE\_FROM\_GROUP, and MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS. MPI\_COMM\_CREATE\_GROUP and MPI\_COMM\_CREATE\_FROM\_GROUP are invoked only by the processes in the group of the new communicator being constructed. MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS is invoked by all the processes in the local and remote groups of the new communicator being constructed. See the discussion below for the definition of local and remote groups.

*Rationale.* Note that, when using the World Model, there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. In the World Model, the base communicator for all MPI communicators is predefined outside of MPI, and is MPI\_COMM\_WORLD. The World Model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

This chapter presents the following communicator construction routines: MPI\_COMM\_CREATE, MPI\_COMM\_DUP, MPI\_COMM\_IDUP, MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_IDUP\_WITH\_INFO, MPI\_COMM\_SPLIT and MPI\_COMM\_SPLIT\_TYPE can be used to create both intra-communicators and intercommunicators; MPI\_COMM\_CREATE\_GROUP, MPI\_COMM\_CREATE\_FROM\_GROUP and MPI\_INTERCOMM\_MERGE (see Section 7.6.2) can be used to create intra-communicators; 

```
1
     MPI_INTERCOMM_CREATE and MPI_INTERCOMM_CREATE_FROM_GROUPS (see Sec-
\mathbf{2}
     tion 7.6.2) can be used to create inter-communicators.
3
          An intra-communicator involves a single group while an inter-communicator involves
4
     two groups. Where the following discussions address inter-communicator semantics, the
\mathbf{5}
     two groups in an inter-communicator are called the left and right groups. A process in an
6
     inter-communicator is a member of either the left or the right group. From the point of
\overline{7}
     view of that process, the group that the process is a member of is called the local group; the
8
     other group (relative to that process) is the remote group. The left and right group labels
9
     give us a way to describe the two groups in an inter-communicator that is not relative to
10
     any particular process (as the local and remote groups are).
11
12
     MPI_COMM_DUP(comm, newcomm)
13
14
       IN
                                              communicator (handle)
                 comm
15
       OUT
                                              copy of comm (handle)
                 newcomm
16
17
     C binding
18
     int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
19
20
     Fortran 2008 binding
21
     MPI_Comm_dup(comm, newcomm, ierror)
22
          TYPE(MPI_Comm), INTENT(IN) :: comm
23
          TYPE(MPI_Comm), INTENT(OUT) :: newcomm
^{24}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     Fortran binding
26
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
27
          INTEGER COMM, NEWCOMM, IERROR
28
29
          MPI_COMM_DUP duplicates the existing communicator comm with associated key
30
     values and topology information. For each key value, the respective copy callback function
^{31}
     determines the attribute value associated with this key in the new communicator; one
32
     particular action that a copy callback may take is to delete the attribute from the new
33
     communicator. MPI_COMM_DUP returns in newcomm a new communicator with the same
34
     group or groups, same topology, and any copied cached information, but a new context (see
35
     Section 7.7.1).
36
37
           Advice to users. This operation is used to provide a parallel library with a duplicate
38
           communication space that has the same properties as the original communicator. This
           includes any attributes (see below) and topologies (see Chapter 8). This call is valid
39
           even if there are pending point-to-point communications involving the communicator
40
41
           comm. A typical call might involve a MPI_COMM_DUP at the beginning of the
42
           parallel call, and an MPI_COMM_FREE of that duplicated communicator at the end
           of the call. Other models of communicator management are also possible.
43
44
           This call applies to both intra- and inter-communicators. (End of advice to users.)
45
46
           Advice to implementors. One need not actually copy the group information, but only
47
           add a new reference and increment the reference count. Copy on write can be used
48
           for the cached information. (End of advice to implementors.)
```

MPI_CON			
IN	comm	communicator (handle)	
IN	info	info object (handle)	
OUT	newcomm	copy of comm (handle)	
001	newcomm	copy of comm (mandle)	
C bindir	ng		
	•	(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)	
Fortran	2008 binding		
		m, info, newcomm, ierror)	
	C(MPI_Comm), INTENT		
TYPE	C(MPI_Info), INTENT	'(IN) :: info	
	C(MPI_Comm), INTENT		
INTE	GER, OPTIONAL, INT	ENT(OUT) :: ierror	
Fortran	binding		
MPI_COMM	L_DUP_WITH_INFO(COM	M, INFO, NEWCOMM, IERROR)	
INTE	GER COMM, INFO, NE	WCOMM, IERROR	
MPI_	_COMM_DUP_WITH_	INFO behaves exactly as MPI_COMM_DUP except that the	
		info are associated with the output communicator newcomm.	
_			
	-	that some hints will only be valid at communicator creation	
an	e. However, for legacy	v reasons, most communicator creation calls do not provide	
time an i	e. However, for legacy nfo argument. One ma	v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator	
time an i	e. However, for legacy nfo argument. One ma	v reasons, most communicator creation calls do not provide	
time an i	e. However, for legacy nfo argument. One ma	v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator	
time an i at c	e. However, for legacy nfo argument. One ma reation time through a	a call to MPI_COMM_DUP_WITH_INFO. ( <i>End of rationale.</i> )	
time an i at c MPI_COM	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo	v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. ( <i>End of rationale.</i> ) comm, request)	
time an i at c MPI_CON IN	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle)</pre>	
time an i at c MPI_CON IN OUT	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, news comm newcomm	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.)</pre>	
time an i at c MPI_CON IN	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle)</pre>	
time an i at c MPI_CON IN OUT OUT	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.)</pre>	
time an i at c MPI_COM IN OUT OUT C bindir	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.)</pre> comm, request) communicator (handle) copy of comm (handle) communication request (handle)	
time an i at c MPI_COM IN OUT OUT C bindin int MPI_	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request ng .Comm_idup(MPI_Comm	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.)</pre>	
tima an i at c MPI_CON IN OUT OUT C bindin int MPI_ Fortran	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request ng Comm_idup(MPI_Comm 2008 binding	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) communication request (handle) communication request (handle) comm, MPI_Comm *newcomm, MPI_Request *request)</pre>	
time an i at c MPI_COM IN OUT OUT C bindir int MPI_ Fortran MPI_Comm	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request ng Comm_idup(MPI_Comm 2008 binding idup(comm, newcom	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) communication request (handle) comm, MPI_Comm *newcomm, MPI_Request *request) m, request, ierror)</pre>	
time an i at c MPI_COM IN OUT OUT C bindir Int MPI_ Fortran IPI_Comm TYPE	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request 10 2008 binding a_idup(comm, newcom C(MPI_Comm), INTENT	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) communication request (handle) d comm, MPI_Comm *newcomm, MPI_Request *request) m, request, ierror) '(IN) :: comm</pre>	
time an i at c MPI_COM IN OUT OUT C bindir int MPI_ Fortran MPI_Comm TYPE TYPE	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request ng Comm_idup(MPI_Comm 2008 binding a_idup(comm, newcom C(MPI_Comm), INTENT C(MPI_Comm), INTENT	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) copy of comm (handle) communication request (handle) d comm, MPI_Comm *newcomm, MPI_Request *request) m, request, ierror) C(IN) :: comm C(OUT), ASYNCHRONOUS :: newcomm</pre>	
time an i at c MPI_CON IN OUT OUT C bindir int MPI_ Fortran MPI_Comm TYPE TYPE TYPE	e. However, for legacy nfo argument. One ma reation time through a /MM_IDUP(comm, newo comm newcomm request 2008 binding a_idup(comm, newcom :(MPI_Comm), INTENT :(MPI_Comm), INTENT :(MPI_Request), INT	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) copy of comm (handle) communication request (handle) d comm, MPI_Comm *newcomm, MPI_Request *request) m, request, ierror) C(IN) :: comm C(OUT), ASYNCHRONOUS :: newcomm ENT(OUT) :: request</pre>	
time an i at c MPI_CON IN OUT OUT C bindir int MPI_ Fortran MPI_Comm TYPE TYPE TYPE INTE	e. However, for legacy nfo argument. One ma reation time through a MM_IDUP(comm, newo comm newcomm request 2008 binding L_idup(comm, newcom C(MPI_Comm), INTENT C(MPI_Comm), INTENT C(MPI_Request), INT CGER, OPTIONAL, INT	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) copy of comm (handle) communication request (handle) d comm, MPI_Comm *newcomm, MPI_Request *request) m, request, ierror) C(IN) :: comm C(OUT), ASYNCHRONOUS :: newcomm</pre>	
time an i at c MPI_COM IN OUT OUT C bindin OUT C bindin int MPI_ Fortran MPI_Comm TYPE TYPE TYPE TYPE INTE	e. However, for legacy nfo argument. One ma reation time through a /MM_IDUP(comm, newo comm newcomm request 2008 binding a_idup(comm, newcom C(MPI_Comm), INTENT C(MPI_Comm), INTENT C(MPI_Request), INT CGER, OPTIONAL, INT binding	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) communication request (handle) d comm, MPI_Comm *newcomm, MPI_Request *request) m, request, ierror) C(IN) :: comm C(OUT), ASYNCHRONOUS :: newcomm ENT(OUT) :: request ENT(OUT) :: ierror</pre>	
time an i at c MPI_COM IN OUT OUT C bindir int MPI_ Fortran MPI_Comm TYPE TYPE INTE Fortran MPI_COMM	e. However, for legacy nfo argument. One ma reation time through a /MM_IDUP(comm, newo comm newcomm request 2008 binding a_idup(comm, newcom C(MPI_Comm), INTENT C(MPI_Comm), INTENT C(MPI_Request), INT CGER, OPTIONAL, INT binding	<pre>v reasons, most communicator creation calls do not provide ay associate info hints with a duplicate of any communicator a call to MPI_COMM_DUP_WITH_INFO. (End of rationale.) communicator (handle) copy of comm (handle) communication request (handle) d comm, MPI_Comm *newcomm, MPI_Request *request) m, request, ierror) '(IN) :: comm '(OUT), ASYNCHRONOUS :: newcomm 'ENT(OUT) :: request 'ENT(OUT) :: ierror M, REQUEST, IERROR)</pre>	

1 2 3 4 5 6 7 8 9	of its nonb was execute after MPI_ assumption MPI_COM It is e	locking behavior, the seman ted at the time that MPI_CC _COMM_IDUP will not be c ns for nonblocking collectiv IM_IDUP and the returned p	nicator <b>newcomm</b> as an input argument to other MPI
11	MPI_COM	IM_IDUP_WITH_INFO(com	m, info, newcomm, request)
12	IN	comm	communicator (handle)
13	IN	info	info object (handle)
14 15	OUT	newcomm	copy of <b>comm</b> (handle)
16	OUT	request	communication request (handle)
17			- 、 ,
18 19	C bindin	g	
20	int MPI_C	-	_Comm comm, MPI_Info info,
21		MPI_Comm *newcomm,	MPI_Request *request)
22		2008 binding	
23		-	nfo, newcomm, request, ierror)
24 25		(MPI_Comm), INTENT(IN) (MPI_Info), INTENT(IN)	
26			, ASYNCHRONOUS :: newcomm
27		(MPI_Request), INTENT(OU	
28	INTEC	GER, OPTIONAL, INTENT(OU	UT) :: ierror
29	Fortran h	oinding	
30 31	MPI_COMM_	_IDUP_WITH_INFO(COMM, I	NFO, NEWCOMM, REQUEST, IERROR)
32	INTEC	GER COMM, INFO, NEWCOMM	, REQUEST, IERROR
33	MPI_	COMM_IDUP_WITH_INFO	is a nonblocking variant of
34			h the exception of its nonblocking behavior, the se-
35			_INFO are as if MPI_COMM_DUP_WITH_INFO was
36 27			_IDUP_WITH_INFO is called. For example, attributes IM_IDUP_WITH_INFO will not be copied to the new
37 38		0	ssumptions for nonblocking collective operations (see
39			DUP_WITH_INFO and the returned request.
40			nicator <b>newcomm</b> as an input argument to other MPI
41	functions l	before the MPI_COMM_IDU	JP_WITH_INFO operation completes.
42	Rati	onale. The MPI_COMM_I	DUP and MPI_COMM_IDUP_WITH_INFO functions
43 44			of purely nonblocking libraries (see [40]). (End of
45		onale.)	
46			
47			
48			

MPI_COMM_CREATE(comm, group, newcomm)			1
IN	comm	communicator (handle)	2
		()	3
IN	group	group, which is a subset of the group of $comm$	4
		(handle)	5
OUT	newcomm	new communicator (handle)	6
			7
C bindir	lo.		8
	0	, MPI_Group group, MPI_Comm *newcomm)	9
1110 III I_		, in i_droup group, in i_oomm (newcomm)	10
Fortran	2008 binding		11
MPI_Comm	_create(comm, group, newco	omm, ierror)	12
TYPE	(MPI_Comm), INTENT(IN) ::	comm	13
TYPE	(MPI_Group), INTENT(IN) :	: group	14
TYPE	(MPI_Comm), INTENT(OUT) :	: newcomm	15
INTE	GER, OPTIONAL, INTENT(OUT)	) :: ierror	16
Fortran	hinding		17
			18

MPI\_COMM\_CREATE(COMM, GROUP, NEWCOMM, IERROR) INTEGER COMM, GROUP, NEWCOMM, IERROR

If comm is an intra-communicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicator. Each process must call MPI\_COMM\_CREATE with a group argument that is a subgroup of the group associated with comm; this could be MPI\_GROUP\_EMPTY. The processes may specify different values for the group argument. If a process calls with a nonempty group then all processes in that group must call the function with the same group as argument, that is the same processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the processes must be disjoint. If the calling process is a member of the group given as group argument, then newcomm is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI\_GROUP\_EMPTY, then MPI\_COMM\_NULL is returned as newcomm. The function is collective and must be called by all processes in the group of comm.

The interface supports the original mechanism from MPI-1.1, which re-Rationale. quired the same group in all processes of comm. It was extended in MPI-2.2 to allow the use of disjoint subgroups in order to allow implementations to eliminate unnecessary communication that MPI\_COMM\_SPLIT would incur when the user already knows the membership of the disjoint subgroups. (*End of rationale.*)

Rationale. The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI\_COMM\_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34 35

36

37

38

39

40 41

42

43 44

45

46

47

1

2

3

4 5

6

7

8

9

10

11

15

16

17

18

19

20

34

35

36

37 38 39

40 41

42

43

44

45

46

47 48 • It permits implementations to sometimes avoid communication related to context creation.

(End of rationale.)

Advice to users. MPI\_COMM\_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI\_COMM\_CREATE, can be used in subsequent calls to MPI\_COMM\_CREATE (or other communicator constructors) to further subdivide a computation into parallel sub-computations. A more general service is provided by MPI\_COMM\_SPLIT, below. (*End of advice to users.*)

Advice to implementors. When calling MPI\_COMM\_DUP, all processes call with the same group (the group associated with the communicator). When calling

MPI\_COMM\_CREATE, the processes provide the same group or disjoint subgroups. For both calls, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system must be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

25If comm is an inter-communicator, then the output communicator is also an inter-com-26municator where the local group consists only of those processes contained in group (see 27Figure 7.1). The group argument should only contain those processes in the local group of 28the input inter-communicator that are to be a part of newcomm. All processes in the same 29 local group of comm must specify the same value for group, i.e., the same members in the 30 same order. If either group does not specify at least one process in the local group of the  $^{31}$ inter-communicator, or if the calling process is not included in the group, MPI\_COMM\_NULL 32 is returned. 33

*Rationale.* In the case where either the left or right group is empty, a null communicator is returned instead of an inter-communicator with MPI\_GROUP\_EMPTY because the side with the empty group must return MPI\_COMM\_NULL. (*End of rationale.*)

**Example 7.1** Inter-communicator creation.

The following example illustrates how the first node in the left side of an inter-communicator could be joined with all members on the right side of an inter-communicator to form a new inter-communicator.

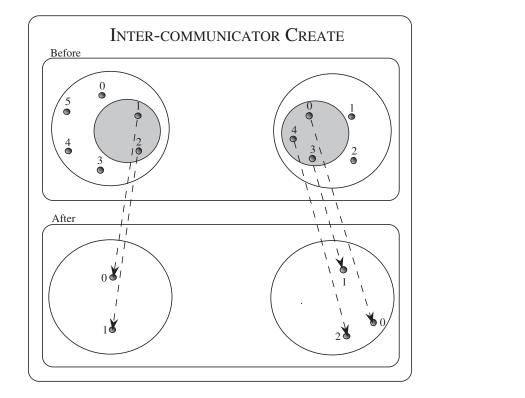


Figure 7.1: Inter-communicator creation using MPI\_COMM\_CREATE extended to intercommunicators. The input groups are those in the grey circle.

```
/* Construct the original inter-communicator: "inter_comm" */
...
/* Construct the group of processes to be in new
    inter-communicator */
if (/* I'm on the left side of the inter-communicator */) {
    MPI_Comm_group(inter_comm, &local_group);
    MPI_Group_incl(local_group, 1, &rank, &group);
    MPI_Group_free(&local_group);
}
else
    MPI_Comm_group(inter_comm, &group);
MPI_Comm_create(inter_comm, group, &new_inter_comm);
MPI_Group_free(&group);
```

 $\mathbf{2}$ 

 $\frac{24}{25}$ 

 $^{31}$ 

334 CHAPTER 7. GROUPS, CONTEXTS, COMMUNICATORS, AND CACHING

MPI\_COMM\_CREATE\_GROUP(comm, group, tag, newcomm)

1

 $\mathbf{2}$ IN intra-communicator (handle) comm 3 IN group, which is a subset of the group of comm group 4(handle) 56 IN tag (integer) tag 7 OUT newcomm new communicator (handle) 8 9 C binding 10 int MPI\_Comm\_create\_group(MPI\_Comm comm, MPI\_Group group, int tag, 11 MPI\_Comm \*newcomm) 1213Fortran 2008 binding 14MPI\_Comm\_create\_group(comm, group, tag, newcomm, ierror) 15TYPE(MPI\_Comm), INTENT(IN) :: comm 16TYPE(MPI\_Group), INTENT(IN) :: group 17INTEGER, INTENT(IN) :: tag 18 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 19INTEGER, OPTIONAL, INTENT(OUT) :: ierror 20Fortran binding 21MPI\_COMM\_CREATE\_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR) 22 INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR 23 $^{24}$ MPI\_COMM\_CREATE\_GROUP is similar to MPI\_COMM\_CREATE; however, 25MPI\_COMM\_CREATE must be called by all processes in the group of comm, whereas 26MPI\_COMM\_CREATE\_GROUP must be called by all processes in group, which is a subgroup 27of the group of comm. In addition, MPI\_COMM\_CREATE\_GROUP requires that comm is 28an intra-communicator. MPI\_COMM\_CREATE\_GROUP returns a new intra-communicator, 29newcomm, for which the group argument defines the communication group. No cached 30 information propagates from comm to newcomm and no virtual topology information is 31added to the created communicator. Each process must provide a group argument that is a 32 subgroup of the group associated with comm; this could be MPI\_GROUP\_EMPTY. If a non-33 empty group is specified, then all processes in that group must call the function, and each of 34these processes must provide the same arguments, including a group that contains the same 35 members with the same ordering. Otherwise the call is erroneous. If the calling process is a 36 member of the group given as the group argument, then newcomm is a communicator with 37 group as its associated group. If the calling process is not a member of group, e.g., group is 38 MPI\_GROUP\_EMPTY, then the call is a local operation and MPI\_COMM\_NULL is returned as 39 newcomm. 4041 Functionality similar to MPI\_COMM\_CREATE\_GROUP can be imple-Rationale. 42mented through repeated MPI\_INTERCOMM\_CREATE and 43 MPI\_INTERCOMM\_MERGE calls that start with the MPI\_COMM\_SELF communicators 44at each process in group and build up an intra-communicator with group group [17]. 45Such an algorithm requires the creation of many intermediate communicators; 46MPI\_COMM\_CREATE\_GROUP can provide a more efficient implementation that avoids 47 this overhead. (End of rationale.) 48

Advice to users. An inter-communicator can be created collectively over processes in the union of the local and remote groups by creating the local communicator using MPI\_COMM\_CREATE\_GROUP and using that communicator as the local communicator argument to MPI\_INTERCOMM\_CREATE. (*End of advice to users.*)

The tag argument does not conflict with tags used in point-to-point communication and is not permitted to be a wildcard. If multiple threads at a given process perform concurrent MPI\_COMM\_CREATE\_GROUP operations, the user must distinguish these operations by providing different tag or comm arguments.

Advice to users. MPI\_COMM\_CREATE may provide lower overhead than MPI\_COMM\_CREATE\_GROUP because it can take advantage of collective communication on comm when constructing newcomm. (*End of advice to users.*)

### MPI\_COMM\_SPLIT(comm, color, key, newcomm)

IN	comm	communicator (handle)
IN	color	control of subset assignment (integer)
IN	key	control of rank assignment (integer)
OUT	newcomm	new communicator (handle)

### C binding

int MPI\_Comm\_split(MPI\_Comm comm, int color, int key, MPI\_Comm \*newcomm)
Fortran 2008 binding
MPI\_Comm\_split(comm, color, key, newcomm, ierror)
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 INTEGER, INTENT(IN) :: color, key
 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortrop binding

# Fortran binding

MPI\_COMM\_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR) INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI\_UNDEFINED, in which case newcomm returns MPI\_COMM\_NULL. This is a collective call, but each process is permitted to provide different values for color and key. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicators.

With an intra-communicator comm, a call to MPI\_COMM\_CREATE(comm, group, newcomm) is equivalent to a call to MPI\_COMM\_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group

1

2

3

4

5

6

 $\overline{7}$ 

8

9 10

11

12

 $^{24}$ 

25 26

27

28

29

30

31

32 33

34

35 36

37

38

39

40

41

42

43

44

45

46

47

(based on a unique numbering of all disjoint groups) and key = rank in group, and all
 processes that are not members of their group argument provide color = MPI\_UNDEFINED.
 The value of color must be non-negative or MPI\_UNDEFINED.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra. For intra-communicators, MPI\_COMM\_SPLIT provides similar capability as MPI\_COMM\_CREATE to split a communicating group into disjoint subgroups. MPI\_COMM\_SPLIT is useful when some processes do not have complete information of the other members in their group, but all processes know (the color of) the group to which they belong. In this case, the MPI implementation discovers the other group members via communication. MPI\_COMM\_CREATE is useful when all processes have complete information of the members of their group. In this case, MPI can avoid the extra communication required to discover group membership. MPI\_COMM\_CREATE\_GROUP is useful when all processes in a given group have complete information of the members of their group and synchronization with processes outside the group can be avoided.

- <sup>20</sup> Multiple calls to MPI\_COMM\_SPLIT can be used to overcome the requirement that <sup>21</sup> any call have no overlap of the resulting communicators (each process is of only one <sup>22</sup> color per call). In this way, multiple overlapping communication structures can be <sup>23</sup> created. Creative use of the color and key in such splitting operations is encouraged.
- Note that, for a fixed color, the keys need not be unique. It is MPI\_COMM\_SPLIT's responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group.
- Essentially, making the key value the same (e.g., zero) for all processes of a given color means that one does not really care about the rank-order of the processes in the new communicator. (*End of advice to users.*)
  - *Rationale.* color is restricted to be non-negative, so as not to conflict with the value assigned to MPI\_UNDEFINED. (*End of rationale.*)
- The result of MPI\_COMM\_SPLIT on an inter-communicator is that those processes on the left with the same color as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the inter-communicator (see Figure 7.2). For those colors that are specified only on one side of the inter-communicator, MPI\_COMM\_NULL is returned. MPI\_COMM\_NULL is also returned to those processes that specify MPI\_UNDEFINED as the color.
  - Advice to users. For inter-communicators, MPI\_COMM\_SPLIT is more general than MPI\_COMM\_CREATE. A single call to MPI\_COMM\_SPLIT can create a set of disjoint inter-communicators, while a call to MPI\_COMM\_CREATE creates only one. (*End of advice to users.*)
- 47 48

43

44

45

46

33

34

35 36

4

5

6

7

8

9

10

11

12

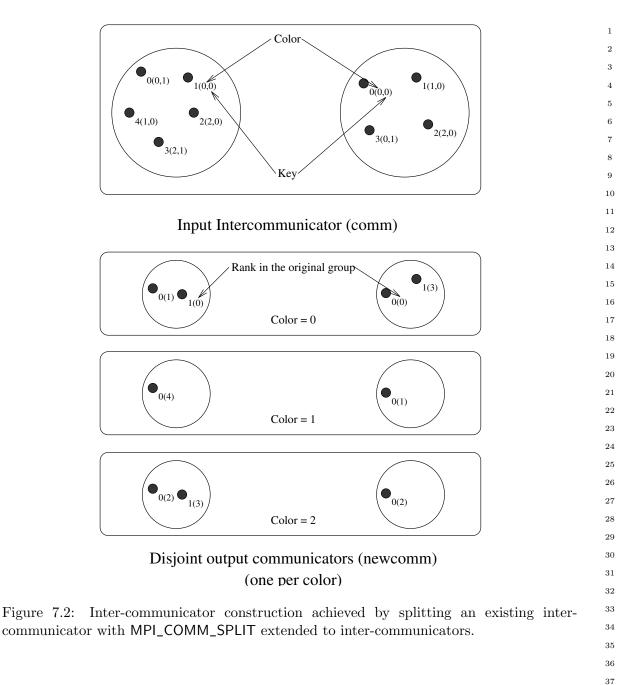
13

14

15

16

17



```
1
     Example 7.2 Parallel client-server model.
\mathbf{2}
     The following client code illustrates how clients on the left side of an inter-communicator
3
     could be assigned to a single server from a pool of servers on the right side of an inter-
4
     communicator.
5
6
              /* Client code */
              MPI_Comm multiple_server_comm;
7
                         single_server_comm;
8
              MPI_Comm
              int
                         color, rank, num_servers;
9
10
11
              /* Create inter-communicator with clients and servers:
                  multiple_server_comm */
12
13
               . . .
14
              /* Find out the number of servers available */
15
16
              MPI_Comm_remote_size(multiple_server_comm, &num_servers);
17
              /* Determine my color */
18
              MPI_Comm_rank(multiple_server_comm, &rank);
19
              color = rank % num_servers;
20
21
              /* Split the inter-communicator */
22
              MPI_Comm_split(multiple_server_comm, color, rank,
23
^{24}
                               &single_server_comm);
25
     The following is the corresponding server code:
26
27
              /* Server code */
28
              MPI_Comm multiple_client_comm;
29
              MPI_Comm single_server_comm;
30
              int
                         rank;
31
32
              /* Create inter-communicator with clients and servers:
33
                  multiple_client_comm */
34
               . . .
35
36
              /* Split the inter-communicator for a single server per group
37
                  of clients */
38
              MPI_Comm_rank(multiple_client_comm, &rank);
39
              MPI_Comm_split(multiple_client_comm, rank, 0,
40
                               &single_server_comm);
41
42
43
44
45
46
```

MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)			1
		,	2
IN	comm	communicator (handle)	3
IN	split_type	type of processes to be grouped together (integer)	4
IN	key	control of rank assignment (integer)	5
INOUT	info	info argument (handle)	6
OUT	newcomm	new communicator (handle)	7 8
			9
C binding	5		10
int MPI_C	<pre>Comm_split_type(MPI_Comm </pre>	comm, int split_type, int key,	11
	MPI_Info info, MPI_C	omm *newcomm)	12
Fortran 2	2008 binding		13
		ype, key, info, newcomm, ierror)	14
	MPI_Comm), INTENT(IN) ::		15
	ER, INTENT(IN) :: split_		16
	<pre>MPI_Info), INTENT(IN) ::</pre>		17
	MPI_Comm), INTENT(OUT) :		18
	ER, OPTIONAL, INTENT(OUT)		19
INIEC	ER, UPIIONAL, INIENI(UUI)	) ielioi	20
Fortran b	binding		21
MPI_COMM_	SPLIT_TYPE(COMM, SPLIT_T	YPE, KEY, INFO, NEWCOMM, IERROR)	22
INTEG	ER COMM, SPLIT_TYPE, KEY	, INFO, NEWCOMM, IERROR	23

INTEGER COMM, SPLIT\_TYPE, KEY, INFO, NEWCOMM, IERROR

This function partitions the group associated with comm into disjoint subgroups such that each subgroup contains all MPI processes in the same grouping referred to by split\_type. Within each subgroup, the MPI processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in **newcomm**. This is a collective call. All MPI processes in the group associated with comm must provide the same split\_type, but each MPI process is permitted to provide different values for key. An exception to this rule is that an MPI process may supply the type value MPI\_UNDEFINED, in which case MPI\_COMM\_NULL is returned in newcomm for such MPI process. No cached information propagates from comm to newcomm and no virtual topology information is added to the created communicators.

For split\_type, the following values are defined by MPI:

MPI\_COMM\_TYPE\_SHARED—all MPI processes in newcomm can create a shared memory segment (e.g., with a successful call to MPI\_WIN\_ALLOCATE\_SHARED). This segment can subsequently be used for load/store accesses by all MPI processes in newcomm.

> Since the location of some of the MPI processes may change Advice to users. during the application execution, the communicators created with the value MPI\_COMM\_TYPE\_SHARED before this change may not reflect an actual ability to share memory between MPI processes after this change. (End of advice to users.)

MPI\_COMM\_TYPE\_HW\_GUIDED—this value specifies that the communicator comm is split according to a hardware resource type (for example a computing core or an L3

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36 37

38

39

40 41

42

43

44

45

4647

1 cache) specified by the "mpi\_hw\_resource\_type" info key. Each output communicator 2 newcomm corresponds to a single instance of the specified hardware resource type. 3 The MPI processes in the group associated with the output communicator newcomm 4 utilize that specific hardware resource type instance, and no other instance of the 5same hardware resource type. 6 If an MPI process does not meet the above criteria, then MPI\_COMM\_NULL is returned 7 in newcomm for such process. 8 MPI\_COMM\_NULL is also returned in **newcomm** in the following cases: 9 10 • MPI\_INFO\_NULL is provided. 11 • The info handle does not include the key "mpi\_hw\_resource\_type". 12• The MPI implementation neither recognizes nor supports the info key 13 14"mpi\_hw\_resource\_type". 15• The MPI implementation does not recognize the value associated with the info 16key "mpi\_hw\_resource\_type". 17 The MPI implementation will return in the group of the output communicator 18 newcomm the largest subset of MPI processes that match the splitting criterion. 19 20The processes in the group associated with newcomm are ranked in the order defined 21by the value of the argument key with ties broken according to their rank in the group 22associated with comm. 23 $^{24}$ Advice to users. The set of hardware resources that an MPI process is able to 25utilize may change during the application execution (e.g., because of the reloca-26tion of an MPI process), in which case the communicators created with the value 27MPI\_COMM\_TYPE\_HW\_GUIDED before this change may not reflect the utiliza-28 tion of hardware resources of such process at any time after the communicator 29 creation. (End of advice to users.) 30 31The user explicitly constrains with the info argument the splitting of the input communicator comm. To this end, the info key "mpi\_hw\_resource\_type" is reserved and 32 33 its associated value is an implementation-defined string designating the type of the 34 requested hardware resource (e.g., "NUMANode", "Package" or "L3Cache"). 35The value "mpi\_shared\_memory" is reserved and its use is equivalent to using 36 MPI\_COMM\_TYPE\_SHARED for the split\_type parameter. 37 38 Rationale. The value "mpi\_shared\_memory" is defined in order to ensure consis-39 tency between the use of MPI\_COMM\_TYPE\_SHARED and the use of 40 MPI\_COMM\_TYPE\_HW\_GUIDED. (*End of rationale.*) 41 All MPI processes must provide the same value for the info key "mpi\_hw\_resource\_type". 4243 MPI\_COMM\_TYPE\_HW\_UNGUIDED—the group of MPI processes associated with newcomm 44must be a *strict* subset of the group associated with comm and each 45newcomm corresponds to a single instance of a hardware resource type (for example 46a computing core or an L3 cache). 47 48

All MPI processes in the group associated with comm which utilize that specific hardware resource type instance—and no other instance of the same hardware resource type—are included in the group of newcomm.

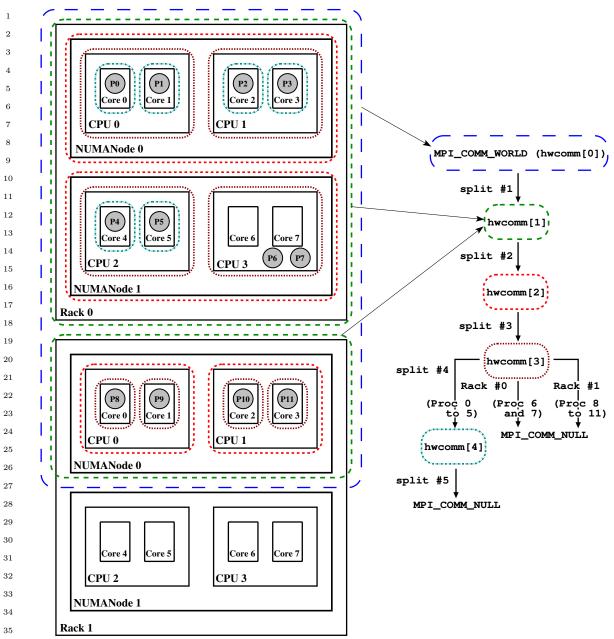
If a given MPI process cannot be a member of a communicator that forms such a strict subset, or does not meet the above criteria, then MPI\_COMM\_NULL is returned in newcomm for this process.

Advice to implementors. In a high-quality MPI implementation, the number of different new valid communicators **newcomm** produced by this splitting operation should be minimal unless the user provides a key/value pair that modifies this behavior. The sets of hardware resource types used for the splitting operation are implementation-dependent, but should reflect the hardware of the actual system on which the application is currently executing. (*End of advice to implementors.*)

*Rationale.* If the hardware resources are hierarchically organized, calling this routine several times using as its input communicator comm the output communicator newcomm of the previous call creates a sequence of newcomm communicators in each MPI process, which exposes a hierarchical view of the hardware platform, as shown in Example 7.4. This sequence of returned newcomm communicators may differ from the sets of hardware resource types, as shown in the second splitting operation in Figure 7.3. (*End of rationale.*)

Advice to users. Each output communicator newcomm can represent a different hardware resource type (see Figure 7.3 for an example). The set of hardware resources an MPI process utilizes may change during the application execution (e.g., because of process relocation), in which case the communicators created with the value MPI\_COMM\_TYPE\_HW\_UNGUIDED before this change may not reflect the utilization of hardware resources for such process at any time after the communicator creation. (*End of advice to users.*)

If a valid info handle is provided as an argument, the MPI implementation sets the info key "mpi\_hw\_resource\_type" for each MPI process in the group associated with a



36

Figure 7.3: Recursive splitting of MPI\_COMM\_WORLD with MPI\_COMM\_SPLIT\_TYPE and 37 MPI\_COMM\_TYPE\_HW\_UNGUIDED. Dashed lines represent communicators whilst solid lines 38 represent hardware resources. MPI processes (P0 to P11) utilize exclusively their respective 39 core, except for P6 and P7 which utilize CPU #3 of Rack #0 and can therefore use Cores 40#6 and #7 indifferently. The second splitting operation yields two subcommunicators  $^{41}$ corresponding to NUMANodes in Rack #0 and to CPUs in Rack #1 because Rack #1 42features only one NUMANode, which corresponds to the whole portion of the Rack that 43 is included in MPI\_COMM\_WORLD and hwcomm[1]. For the first splitting operation, the 44hardware resource type returned in the info argument is "Rack" on the processes on Rack 45#0, whereas on Rack #1, it can be either "Rack" or "NUMANode". 46

returned **newcomm** communicator and the info key value is an implementation-defined string that indicates the hardware resource type represented by **newcomm**. The same hardware resource type must be set in all MPI processes in the group associated with **newcomm**.

```
Example 7.4 Recursive splitting of MPI_COMM_WORLD.
  #define MAX_NUM_LEVELS 32
 MPI_Comm hwcomm[MAX_NUM_LEVELS];
  int
           rank, level_num = 0;
 hwcomm[level_num] = MPI_COMM_WORLD;
 while((hwcomm[level_num] != MPI_COMM_NULL)
        && (level_num < MAX_NUM_LEVELS-1)){
   MPI_Comm_rank(hwcomm[level_num],&rank);
    MPI_Comm_split_type(hwcomm[level_num],
                        MPI_COMM_TYPE_HW_UNGUIDED,
                        rank,
                        MPI_INFO_NULL,
                        &hwcomm[level_num+1]);
    level_num++;
  }
```

Advice to implementors. Implementations can define their own split\_type values, or use the info argument, to assist in creating communicators that help expose platformspecific information to the application. The concept of hardware-based communicators was first described by Träff [67] for SMP systems. Guided and unguided modes description as well as an implementation path are introduced by Goglin *et al.* [27]. (*End of advice to implementors.*)

MPI_COMM_CREATE_FROM_GROUP(group, stringtag, info, errhandler, newcomm)				
IN	group	group (handle)		
IN	stringtag	unique identifier for this operation (string)		
IN	info	info object (handle)		
IN	errhandler	error handler to be attached to new		
		intra-communicator (handle)		
OUT	newcomm	new communicator (handle)		
C binding				

int MPI\_Comm\_create\_from\_group(MPI\_Group group, const char \*stringtag, MPI\_Info info, MPI\_Errhandler errhandler, MPI\_Comm \*newcomm)  $\mathbf{2}$ 

1 Fortran 2008 binding  $\mathbf{2}$ MPI\_Comm\_create\_from\_group(group, stringtag, info, errhandler, newcomm, 3 ierror) 4 TYPE(MPI\_Group), INTENT(IN) :: group 5CHARACTER(LEN=\*), INTENT(IN) :: stringtag 6 TYPE(MPI\_Info), INTENT(IN) :: info 7TYPE(MPI\_Errhandler), INTENT(IN) :: errhandler 8 TYPE(MPI\_Comm), INTENT(OUT) :: newcomm 9 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 10 Fortran binding 11 MPI\_COMM\_CREATE\_FROM\_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM, 12IERROR) 13 INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR 14CHARACTER\*(\*) STRINGTAG 1516MPI\_COMM\_CREATE\_FROM\_GROUP is similar to MPI\_COMM\_CREATE\_GROUP, ex-17cept that the set of MPI processes involved in the creation of the new intra-communicator 18is specified by a group argument, rather than the group associated with a pre-existing com-19municator. If a non-empty group is specified, then all MPI processes in that group must call 20the function and each of these MPI processes must provide the same arguments, including 21a group that contains the same members with the same ordering, and identical stringtag 22value. In the event that MPI\_GROUP\_EMPTY is supplied as the group argument, then the 23call is a local operation and MPI\_COMM\_NULL is returned as newcomm. The stringtag argu-24ment is analogous to the tag used for MPI\_COMM\_CREATE\_GROUP. If multiple threads at 25a given MPI process perform concurrent MPI\_COMM\_CREATE\_FROM\_GROUP operations, 26the user must distinguish these operations by providing different stringtag arguments. The 27stringtag shall not exceed MPI\_MAX\_STRINGTAG\_LEN characters in length. For C, this in-28 cludes space for a null terminating character. MPI\_MAX\_STRINGTAG\_LEN shall have a value 29of at least 63. 30 The errhandler argument specifies an error handler to be attached to the new intra- $^{31}$ communicator. This error handler will also be invoked if the 32 MPI\_COMM\_CREATE\_FROM\_GROUP function encounters an error. The 33 info argument provides hints and assertions, possibly MPI implementation dependent, which 34indicate desired characteristics and guide communicator creation. 35 36 Advice to users. The stringtag argument is used to distinguish concurrent commu-37 nicator construction operations issued by different entities. As such, it is important 38 to ensure that this argument is unique for each concurrent call to MPI\_COMM\_CREATE\_FROM\_GROUP. Reverse domain name notation convention [1] 39 40 is one approach to constructing unique stringtag arguments. See also example 11.9. 41 (End of advice to users.) 4243 44454647

7.4.3 Communicator Destructors	
MPI_COMM_FREE(comm) INOUT comm	communicator to be destroyed (handle)
C binding int MPI_Comm_free(MPI_Comm *comm)	
Fortran 2008 binding MPI_Comm_free(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) INTEGER, OPTIONAL, INTENT(OUT)	
Fortran binding MPI_COMM_FREE(COMM, IERROR) INTEGER COMM, IERROR	

This collective operation marks the communication object for deallocation. The handle is set to MPI\_COMM\_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 7.7) are called in arbitrary order.

Advice to implementors. Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

#### 7.4.4 Communicator Info

Hints specified via info (see Chapter 10) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or minimize use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI\_COMM\_GET\_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per communicator basis, in MPI\_COMM\_DUP\_WITH\_INFO, MPI\_COMM\_IDUP\_WITH\_INFO, MPI\_COMM\_SET\_INFO, MPI\_COMM\_SPLIT\_TYPE, MPI\_DIST\_GRAPH\_CREATE, and MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI\_COMM\_SET\_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

 $^{31}$ 

Info hints are not propagated by MPI from one communicator to another. The following  $^{2}$ info keys are valid for all communicators.

- 4 "mpi\_assert\_no\_any\_tag" (boolean, default: "false"): If set to "true", then the implementation may assume that the process will not use the MPI\_ANY\_TAG wildcard on the 56 given communicator.
- "mpi\_assert\_no\_any\_source" (boolean, default: "false"): If set to "true", then the implementation may assume that the process will not use the MPI\_ANY\_SOURCE wildcard on the given communicator. 10
  - "mpi\_assert\_exact\_length" (boolean, default: "false"): If set to "true", then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.
- "mpi\_assert\_allow\_overtaking" (boolean, default: "false"): If set to "true", then the im-16plementation may assume that point-to-point communications on the given commu-17 nicator do not rely on the non-overtaking rule specified in Section 3.5. In other words, 18 the application asserts that send operations are not required to be matched at the 19 receiver in the order in which the send operations were posted by the sender, and 20receive operations are not required to be matched in the order in which they were 21posted by the receiver. 22
  - Advice to users. Use of the "mpi\_assert\_allow\_overtaking" info key can result in nondeterminism in the message matching order. (End of advice to users.)
  - Advice to users. Some optimizations may only be possible when all processes in the group of the communicator provide a given info key with the same value. (End of advice to users.)

32 MPI\_COMM\_SET\_INFO(comm, info)

1

3

 $\overline{7}$ 

8

9

11

12

13

14

15

23

24

2526

27

28

```
33
       INOUT
                comm
                                           communicator (handle)
34
       IN
                info
                                           info object (handle)
35
36
37
     C binding
38
     int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
39
     Fortran 2008 binding
40
     MPI_Comm_set_info(comm, info, ierror)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Info), INTENT(IN) :: info
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     Fortran binding
46
     MPI_COMM_SET_INFO(COMM, INFO, IERROR)
47
         INTEGER COMM, INFO, IERROR
48
```

MPI\_COMM\_SET\_INFO updates the hints of the communicator associated with comm using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI\_COMM\_SET\_INFO. MPI\_COMM\_SET\_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

Advice to users. Some info items that an implementation can use when it creates a communicator cannot easily be changed once the communicator has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI\_COMM\_SET\_INFO. MPI\_COMM\_GET\_INFO can be used to determine whether updates to existing info hints were ignored by the implementation. (*End of advice to users.*)

Advice to users. Setting info hints on the predefined communicators MPI\_COMM\_WORLD and MPI\_COMM\_SELF may have unintended effects, as changes to these global objects may affect all components of the application, including libraries and tools. Users must ensure that all components of the application that use a given communicator, including libraries and tools, can comply with any info hints associated with that communicator. (*End of advice to users.*)

# MPI\_COMM\_GET\_INFO(comm, info\_used)

IN	comm	communicator object (handle)
OUT	info_used	new info object (handle)

#### C binding

int MPI\_Comm\_get\_info(MPI\_Comm comm, MPI\_Info \*info\_used)

# Fortran 2008 binding

MPI\_Comm\_get\_info(comm, info\_used, ierror)
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 TYPE(MPI\_Info), INTENT(OUT) :: info\_used
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

# Fortran binding

MPI\_COMM\_GET\_INFO(COMM, INFO\_USED, IERROR) INTEGER COMM, INFO\_USED, IERROR

41 MPI\_COMM\_GET\_INFO returns a new info object containing the hints of the commu-42nicator associated with comm. The current setting of all hints related to this communicator is returned in info\_used. An MPI implementation is required to return all hints that are 4344supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by 4546the implementation. If no such hints exist, a handle to a newly created info object is re-47turned that contains no key/value pair. The user is responsible for freeing info\_used via 48 MPI\_INFO\_FREE.

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9

10

11

12

13

14

15 16

17

18

19

20

21

22 23 24

25 26

27 28 29

30

31

32 33

34

35

36 37

38

39

7.5 Motivating Examples

7.5.1 Current Practice #1

```
Example 7.5 Parallel output of a message
int main(int argc, char *argv[])
{
    int me, size;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    (void)printf("Process %d size %d\n", me, size);
    ...
    MPI_Finalize();
    return 0;
}
```

Example 7.5 is a do-nothing program that initializes itself, and refers to the "all" communicator, and prints a message. It terminates itself too. This example does not imply that MPI supports printf-like communication itself.

```
Example 7.6 Message exchange (supposing that size is even)
    int main(int argc, char *argv[])
    {
       int me, size;
       int SOME_TAG = 0;
       . . .
       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &me);
                                             /* local */
       MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
       if((me % 2) == 0)
       ſ
          /* send unless highest-numbered process */
          if((me + 1) < size)
             MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
       }
       else
          MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
       . . .
       MPI_Finalize();
       return 0;
```

```
}
Example 7.6 schematically illustrates message exchanges between "even" and "odd" pro-
cesses in the "all" communicator.
7.5.2 Current Practice #2
Example 7.7
   int main(int argc, char *argv[])
   {
     int me, count;
     void *data;
     . . .
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &me);
     if(me == 0)
     {
          /* get input, create buffer ''data'' */
          . . .
     }
     MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);
     . . .
     MPI_Finalize();
     return 0;
   }
```

Example 7.7 illustrates the use of a collective communication.

7.5.3 (Approximate) Current Practice #3

```
Example 7.8
```

```
int main(int argc, char *argv[])
{
    int me, count, count2;
    void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
    MPI_Group group_world, grprem;
    MPI_Comm commWorker;
    static int ranks[] = {0};
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
```

 $\mathbf{2}$ 

```
MPI_Group_excl(group_world, 1, ranks, &grprem); /* local */
  MPI_Comm_create(MPI_COMM_WORLD, grprem, &commWorker);
  if(me != 0)
  {
    /* compute on worker */
    MPI_Reduce(send_buf,recv_buf,count, MPI_INT, MPI_SUM, 1, commWorker);
    . . .
    MPI_Comm_free(&commWorker);
  }
  /* zero falls through immediately to this reduce, others do later... */
  MPI_Reduce(send_buf2, recv_buf2, count2,
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Group_free(&group_world);
  MPI_Group_free(&grprem);
  MPI_Finalize();
  return 0;
}
```

Example 7.8 illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commWorker) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI\_COMM\_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI\_COMM\_WORLD is insulated from communicators because distinct contexts within communicators because distinct contexts within communicators are enforced to be unique on any process.

# 7.5.4 Communication Safety Example

The following example (7.9) is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

# Example 7.9 #define TAG\_A

 $^{24}$ 

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT 50
int main(int argc, char *argv[])
{
    int me;
    MPI_Request request[2];
    MPI_Status status[2];
    MPI_Group group_world, subgroup;
    int ranks[] = {2, 4, 6, 8};
```

```
1
     MPI_Comm the_comm;
                                                                                        \mathbf{2}
      . . .
                                                                                        3
     MPI_Init(&argc, &argv);
                                                                                        4
     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
                                                                                        5
                                                                                        6
     MPI_Group_incl(group_world, 4, ranks, &subgroup); /* local */
                                                                                        7
     MPI_Group_rank(subgroup, &me);
                                            /* local */
                                                                                        9
     MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
                                                                                        10
                                                                                        11
      if(me != MPI_UNDEFINED)
                                                                                        12
      {
                                                                                        13
          MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                                                                                        14
                              the_comm, request);
                                                                                        15
          MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                                                                                        16
                              the_comm, request+1);
                                                                                        17
          for(i = 0; i < SOME_COUNT; i++)</pre>
                                                                                        18
            MPI_Reduce(..., the_comm);
                                                                                        19
          MPI_Waitall(2, request, status);
                                                                                        20
                                                                                        21
          MPI_Comm_free(&the_comm);
                                                                                        22
     }
                                                                                        23
                                                                                        ^{24}
     MPI_Group_free(&group_world);
                                                                                        25
     MPI_Group_free(&subgroup);
                                                                                        26
     MPI_Finalize();
                                                                                        27
     return 0;
                                                                                        28
   }
                                                                                        29
                                                                                        30
      Library Example \#1
7.5.5
                                                                                        31
The main program:
                                                                                        32
                                                                                        33
   int main(int argc, char *argv[])
                                                                                        34
   {
                                                                                        35
     int done = 0;
                                                                                        36
     user_lib_t *libh_a, *libh_b;
                                                                                        37
     void *dataset1, *dataset2;
                                                                                        38
     . . .
                                                                                        39
     MPI_Init(&argc, &argv);
                                                                                        40
     . . .
                                                                                        41
     init_user_lib(MPI_COMM_WORLD, &libh_a);
                                                                                        42
```

```
init_user_lib(MPI_COMM_WORLD, &libh_b);
                                                                                    43
. . .
                                                                                     44
user_start_op(libh_a, dataset1);
                                                                                     45
user_start_op(libh_b, dataset2);
                                                                                     46
. . .
                                                                                     47
while(!done)
                                                                                     48
```

```
1
           {
\mathbf{2}
              /* work */
3
               . . .
4
              MPI_Reduce(..., MPI_COMM_WORLD);
5
               . . .
6
              /* see if done */
7
               . . .
8
           }
9
           user_end_op(libh_a);
10
           user_end_op(libh_b);
11
12
           uninit_user_lib(libh_a);
13
           uninit_user_lib(libh_b);
14
           MPI_Finalize();
15
           return 0;
16
        }
17
     The user library initialization code:
18
19
        void init_user_lib(MPI_Comm comm, user_lib_t **handle)
20
         {
21
           user_lib_t *save;
22
23
           user_lib_initsave(&save); /* local */
^{24}
           MPI_Comm_dup(comm, &(save->comm));
25
26
           /* other inits */
27
           . . .
28
29
           *handle = save;
30
        }
31
32
     User start-up code:
33
        void user_start_op(user_lib_t *handle, void *data)
34
         {
35
           MPI_Irecv( ..., handle->comm, &(handle->irecv_handle) );
36
           MPI_Isend( ..., handle->comm, &(handle->isend_handle) );
37
        }
38
39
     User communication clean-up code:
40
41
        void user_end_op(user_lib_t *handle)
42
         {
           MPI_Status status;
43
44
           MPI_Wait(&handle->isend_handle, &status);
45
           MPI_Wait(&handle->irecv_handle, &status);
46
        }
47
     User object clean-up code:
48
```

```
1
   void uninit_user_lib(user_lib_t *handle)
                                                                                       \mathbf{2}
   {
                                                                                       3
     MPI_Comm_free(&(handle->comm));
     free(handle);
                                                                                       4
   }
                                                                                       5
                                                                                       6
                                                                                       7
7.5.6 Library Example \#2
                                                                                       8
The main program:
                                                                                       9
                                                                                       10
   int main(int argc, char *argv[])
                                                                                       11
   {
                                                                                       12
     int ma, mb;
                                                                                       13
     MPI_Group group_world, group_a, group_b;
                                                                                       14
     MPI_Comm comm_a, comm_b;
                                                                                       15
                                                                                       16
     static int list_a[] = \{0, 1\};
                                                                                       17
#if defined(EXAMPLE_2B) || defined(EXAMPLE_2C)
                                                                                       18
     static int list_b[] = {0, 2,3};
                                                                                       19
#else/* EXAMPLE_2A */
                                                                                       20
     static int list_b[] = \{0, 2\};
                                                                                       21
#endif
                                                                                       22
     int size_list_a = sizeof(list_a)/sizeof(int);
                                                                                       23
     int size_list_b = sizeof(list_b)/sizeof(int);
                                                                                       ^{24}
                                                                                       25
     . . .
                                                                                       26
     MPI_Init(&argc, &argv);
                                                                                       27
     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
                                                                                       28
                                                                                       29
     MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
                                                                                       30
     MPI_Group_incl(group_world, size_list_b, list_b, &group_b);
                                                                                       ^{31}
                                                                                       32
     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
                                                                                       33
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);
                                                                                       34
                                                                                       35
     if(comm_a != MPI_COMM_NULL)
                                                                                       36
        MPI_Comm_rank(comm_a, &ma);
                                                                                       37
     if(comm_b != MPI_COMM_NULL)
                                                                                       38
        MPI_Comm_rank(comm_b, &mb);
                                                                                       39
                                                                                       40
     if(comm_a != MPI_COMM_NULL)
                                                                                       41
        lib_call(comm_a);
                                                                                       42
                                                                                       43
     if(comm_b != MPI_COMM_NULL)
                                                                                       44
     ſ
                                                                                       45
       lib_call(comm_b);
                                                                                       46
       lib_call(comm_b);
                                                                                       47
     }
                                                                                       48
```

```
1
2
           if(comm_a != MPI_COMM_NULL)
3
             MPI_Comm_free(&comm_a);
4
           if(comm_b != MPI_COMM_NULL)
5
             MPI_Comm_free(&comm_b);
6
           MPI_Group_free(&group_a);
7
           MPI_Group_free(&group_b);
8
           MPI_Group_free(&group_world);
9
           MPI_Finalize();
10
           return 0;
11
        }
12
     The library:
13
         void lib_call(MPI_Comm comm)
14
        ſ
15
           int me, done = 0;
16
           MPI_Status status;
17
           MPI_Comm_rank(comm, &me);
18
           if(me == 0)
19
              while(!done)
20
              {
21
                  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
22
                  . . .
23
              }
24
           else
25
           {
26
             /* work */
27
             MPI_Send(..., 0, ARBITRARY_TAG, comm);
28
             . . .
29
           }
30
     #ifdef EXAMPLE_2C
^{31}
           /* include (resp, exclude) for safety (resp, no safety): */
32
           MPI_Barrier(comm);
33
     #endif
34
        }
35
```

The above example is really three examples, depending on whether or not one includes rank 3 in list\_b, and whether or not a synchronize is included in lib\_call. This example illustrates that, despite contexts, subsequent calls to lib\_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI\_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back-masking.

<sup>43</sup> Algorithms like "reduce" and "allreduce" have strong enough source selectivity prop-<sup>44</sup> erties so that they are inherently okay (no back-masking), provided that MPI provides basic <sup>45</sup> guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root <sup>46</sup> or different roots (see [64]). Here we rely on two guarantees of MPI: pairwise ordering <sup>47</sup> of messages between processes in the same context, and source selectivity—deleting either <sup>48</sup> feature removes the guarantee that back-masking cannot be required. Algorithms that try to do nondeterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of "reduce," "allreduce," and "broadcast." Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of "collective calls" implemented with point-topoint operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 7.9.

# 7.6 Inter-Communication

users.)

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called "intra-communication" and the communicator used is called an "intra-communicator," as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called "inter -communication" and the communicator used is called an "inter-communicator," as introduced earlier.

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the "local group," that is, the sender in a send and the receiver in a receive. The group containing the target process is called the "remote group," that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking except for MPI\_COMM\_IDUP and require that the local and remote groups be disjoint.

Advice to users. The groups must be disjoint for several reasons. Primarily, this is the intent of the inter-communicators—to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI\_INTERCOMM\_MERGE, which allows the user to control the ranking of the processes in the created intra-communicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to inter-communicators makes the most sense when the groups are disjoint. (*End of advice to* 

Here is a summary of the properties of inter-communication and inter-communicators:

1

 $\mathbf{2}$ 

3 4

 $\mathbf{5}$ 

6 7

8 9 10

11 12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35

36

37

38 39

40 41

42

43

44

 $45 \\ 46$ 

47

1 2 3	• The syntax of point-to-point and collective communication is the same for both inter- and intra-communication. The same communicator can be used both for send and for receive operations.
4 5 6	• A target process is addressed by its rank in the remote group, both for sends and for receives.
7 8 9	• Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
10	• A communicator will provide either intra- or inter-communication, never both.
11 12 13 14 15	The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_CART_CREATE).
16 17 18	Advice to implementors. For the purpose of point-to-point communication, commu- nicators can be represented in each process by a tuple consisting of:
19	group
20	send_context
21	receive_context
22 23	source
24 25 26 27 28	For inter-communicators, group describes the remote group, and source is the rank of the process in the local group. For intra-communicators, group is the communicator group (remote=local), source is the rank of the process in this group, and send context and receive context are identical. A group can be represented by a rank-to-absolute-address translation table.
29 30 31 32 33	The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process $\mathbf{P}$ in group $\mathcal{P}$ , which has an inter-communicator $\mathbf{C}_{\mathcal{P}}$ , and a process $\mathbf{Q}$ in group $\mathcal{Q}$ , which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$ . Then
34	• $\mathbf{C}_{\mathcal{P}}$ .group describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{O}}$ .group describes the group $\mathcal{P}$ .
35 36	• $C_{\mathcal{P}}$ .send_context = $C_{\mathcal{Q}}$ .receive_context and the context is unique in $\mathcal{Q}$ ; $C_{\mathcal{P}}$ .receive_context = $C_{\mathcal{Q}}$ .send_context and this context is unique in $\mathcal{P}$ .
37	• $\mathbf{C}_{\mathcal{P}}$ .source is rank of <b>P</b> in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$ .source is rank of <b>Q</b> in $\mathcal{Q}$ .
38	
39 40 41	Assume that <b>P</b> sends a message to <b>Q</b> using the inter-communicator. Then <b>P</b> uses the <b>group</b> table to find the absolute address of <b>Q</b> ; <b>source</b> and <b>send_context</b> are appended to the message.
42 43 44	Assume that <b>Q</b> posts a receive with an explicit source argument using the inter- communicator. Then <b>Q</b> matches <b>receive_context</b> to the message context and source argument to the message source.
45 46	The same algorithm is appropriate for intra-communicators as well.
47 48	In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the

local communication group, and additional safe contexts. (*End of advice to implementors.*)

### 7.6.1 Inter-Communicator Accessors

### MPI\_COMM\_TEST\_INTER(comm, flag)

IN	comm	communicator (handle)
OUT	flag	true if comm is an inter-communicator (logical)

#### C binding

int	MPI_Comm_tes	_inter(MPI_Com	m comm,	int	*flag)	

### Fortran 2008 binding

<pre>MPI_Comm_test_inter(comm, flag, ierror)</pre>
TYPE(MPI_Comm), INTENT(IN) :: comm
LOGICAL, INTENT(OUT) :: flag
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

### Fortran binding

```
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
INTEGER COMM, IERROR
LOGICAL FLAG
```

This local routine allows the calling process to determine if a communicator is an intercommunicator or an intra-communicator. It returns true if it is an inter-communicator, otherwise false.

When an inter-communicator is used as an input argument to the communicator accessors described above under intra-communication, the following table describes behavior.

MPI_COMM_SIZE	returns the size of the local group.
MPI_COMM_GROUP	returns the local group.
MPI_COMM_RANK	returns the rank in the local group

 Table 7.1: MPI\_COMM\_\* Function Behavior (in Inter-Communication Mode)

Furthermore, the operation MPI\_COMM\_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI\_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI\_CONGRUENT or MPI\_SIMILAR. In particular, it is possible for MPI\_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an intercommunicator. The following are all local operations.  $\mathbf{2}$ 

 $^{24}$ 

```
1
     MPI_COMM_REMOTE_SIZE(comm, size)
\mathbf{2}
       IN
                                            inter-communicator (handle)
                 comm
3
       OUT
                size
                                            number of processes in the remote group of comm
4
                                            (integer)
5
6
     C binding
7
     int MPI_Comm_remote_size(MPI_Comm comm, int *size)
8
9
     Fortran 2008 binding
10
     MPI_Comm_remote_size(comm, size, ierror)
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         INTEGER, INTENT(OUT) :: size
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
17
         INTEGER COMM, SIZE, IERROR
18
19
20
     MPI_COMM_REMOTE_GROUP(comm, group)
21
       IN
                                            inter-communicator (handle)
                comm
22
       OUT
                                            remote group corresponding to comm (handle)
23
                group
^{24}
25
     C binding
26
     int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
27
     Fortran 2008 binding
28
     MPI_Comm_remote_group(comm, group, ierror)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         TYPE(MPI_Group), INTENT(OUT) :: group
^{31}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     Fortran binding
34
     MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
35
          INTEGER COMM, GROUP, IERROR
36
37
          Rationale.
                        Symmetric access to both the local and remote groups of an inter-
38
          communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE
39
          have been provided. (End of rationale.)
40
41
     7.6.2 Inter-Communicator Operations
42
43
     This section introduces five blocking inter-communicator operations.
44
     MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-com-
45
     municator; the function MPI_INTERCOMM_CREATE_FROM_GROUPS constructs an inter-
46
     communicator from two previously defined disjoint groups; the function
47
     MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote
48
```

groups of an inter-communicator. The functions MPI\_COMM\_DUP and MPI\_COMM\_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock.

The function MPI\_INTERCOMM\_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the "group leader") has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator. Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

When using the World Model (Section 11.2), the MPI\_COMM\_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that use the Sessions Model, or the spawn or join operations, it may be necessary to first create an intra-communicator to be used as the peer communicator.

The application topology functions described in Chapter 8 do not apply to intercommunicators. Users that require this capability should utilize MPI\_INTERCOMM\_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topologyoriented intra-communicator. Alternatively, it may be reasonable to devise one's own application topology mechanisms for this case, without loss of generality.

# MPI\_INTERCOMM\_CREATE(local\_comm, local\_leader, peer\_comm, remote\_leader, tag, newintercomm)

IN	local_comm	local intra-communicator (handle)	28
IN	local_leader	rank of local group leader in local_comm (integer)	29
IN	peer_comm	"peer" communicator; significant only at the	30 31
	Poor _ oo	local_leader (handle)	32
IN	remote_leader	rank of remote group leader in peer_comm;	33
		significant only at the local_leader (integer)	34
IN	tag	tag (integer)	35 36
OUT	newintercomm	new inter-communicator (handle)	36 37

#### C binding

<pre>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,</pre>	
MPI_Comm peer_comm, int remote_leader, int tag,	
MPI_Comm *newintercomm)	

### Fortran 2008 binding

MPI\_Intercomm\_create(local\_comm, local\_leader, peer\_comm, remote\_leader, tag, newintercomm, ierror) TYPE(MPI\_Comm), INTENT(IN) :: local\_comm, peer\_comm INTEGER, INTENT(IN) :: local\_leader, remote\_leader, tag

1 2

3

4

5

6 7

8 9

10 11

12

13

14 15

16

17

18

19

20

21

22

23 24 25

26

27

38

39

1 2		MPI_Comm), INTENT(OUT) : ER, OPTIONAL, INTENT(OUT)				
3 4 5 6	Fortran binding MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG, NEWINTERCOMM, IERROR)					
7 8	INTEG	ER LOCAL_COMM, LOCAL_LEAN NEWINTERCOMM, IERRON	DER, PEER_COMM, REMOTE_LEADER, TAG, R			
9 10 11 12 13	This call creates an inter-communicator. It is collective over the union of the local and remote groups. MPI processes should provide identical local_comm and local_leader arguments within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.					
14 15 16	MPI_INTE		ROUPS(local_group, local_leader, remote_group, info, errhandler, newintercomm)			
17	IN	local_group	local group (handle)			
18 19	IN	local_leader	rank of local group leader in $local\_group$ (integer)			
20	IN	remote_group	remote group, significant only at $local\_leader$ (handle)			
21 22	IN	remote_leader	rank of remote group leader in <code>remote_group</code> , significant only at <code>local_leader</code> (integer)			
23 24	IN	stringtag	unique idenitifier for this operation (string)			
25	IN	info	info object (handle)			
26 27	IN	errhandler	error handler to be attached to new inter-communicator (handle)			
28 29	OUT	newintercomm	new inter-communicator (handle)			
30 31 32 33 34 35	C binding int MPI_I	ntercomm_create_from_grou int local_leader, MP const char *stringta	ups(MPI_Group local_group, I_Group remote_group, int remote_leader, g, MPI_Info info, ndler, MPI_Comm *newintercomm)			
36	Fortran 2	2008 binding				
37 38	MPI_Inter		local_group, local_leader, remote_group,			
39		remote_leader, strin ierror)	gtag, info, errhandler, newintercomm,			
40	TYPE(		: local_group, remote_group			
41		ER, INTENT(IN) :: local_				
42 43		CTER(LEN=*), INTENT(IN)	5 5			
44		MPI_Info), INTENT(IN) :: MPI_Errhandler), INTENT(I				
45		MPI_Comm), INTENT(OUT) :				
46		ER, OPTIONAL, INTENT(OUT)				
47 48						

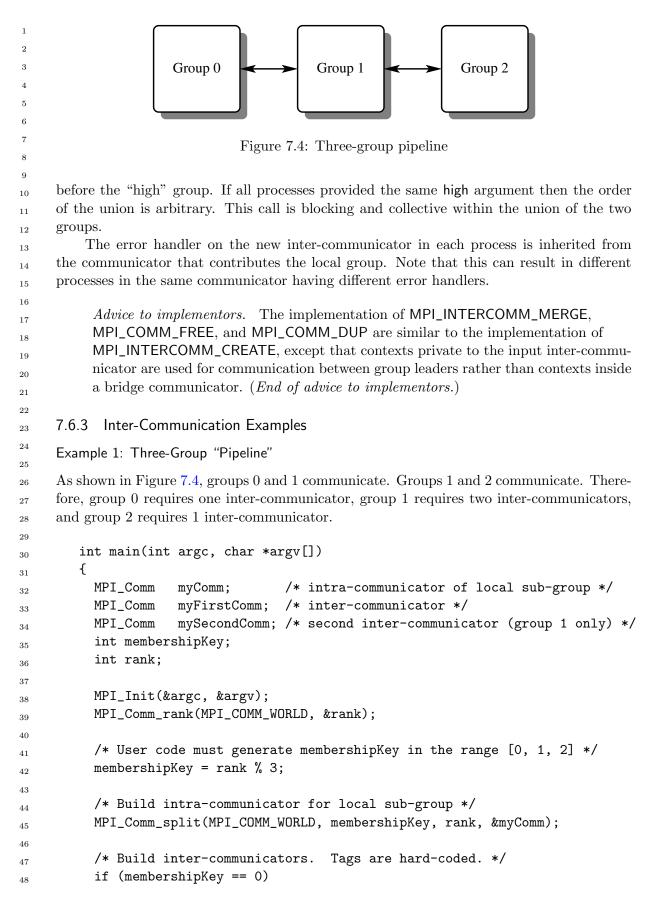
## Fortran binding MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS(LOCAL\_GROUP, LOCAL\_LEADER, REMOTE\_GROUP, REMOTE\_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM, IERROR) INTEGER LOCAL\_GROUP, LOCAL\_LEADER, REMOTE\_GROUP, REMOTE\_LEADER, INFO, ERRHANDLER, NEWINTERCOMM, IERROR CHARACTER\*(\*) STRINGTAG

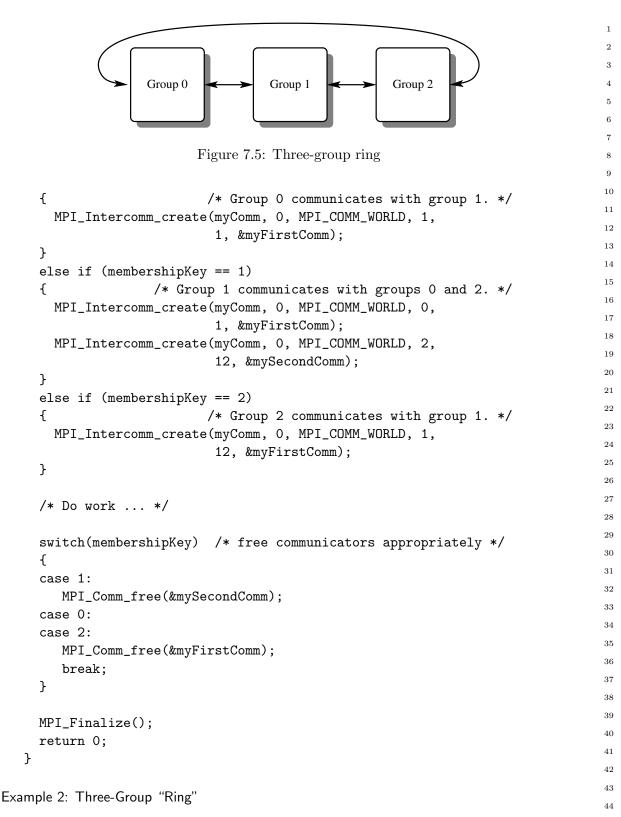
This call creates an inter-communicator. Unlike MPI\_INTERCOMM\_CREATE, this function uses as input previously defined, disjoint local and remote groups. The calling MPI process must be a member of the local group. The call is collective over the union of the local and remote groups. All involved MPI processes shall provide an identical value for the stringtag argument. Within each group, all MPI processes shall provide identical local\_group, local\_leader arguments. Wildcards are not permitted for the remote\_leader or local\_leader arguments. The stringtag argument serves the same purpose as the stringtag used in the MPI\_COMM\_CREATE\_FROM\_GROUP function; it differentiates concurrent calls in a multithreaded environment. The stringtag shall not exceed MPI\_MAX\_STRINGTAG\_LEN characters in length. For C, this includes space for a null terminating character. MPI\_MAX\_STRINGTAG\_LEN shall have a value of at least 63. In the event that MPI\_GROUP\_EMPTY is supplied as the local\_group or remote\_group or both, then the call is a local operation and MPI\_COMM\_NULL is returned as the newintercomm.

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)			2	
	× ×	<b>-</b> ,	2	
IN	intercomm	inter-communicator (handle)	2	
IN	high	ordering of the local and remote groups in the new	2	
		intra-communicator (logical)	2	
OUT	newintracomm	new intra-communicator (handle)	2	
			2	
C hinding	*		3	
	C binding			
<pre>int MPI_Intercomm_merge(MPI_Comm intercomm, int high,</pre>			3	
	MPI_Comm *newintracor	nm)	з	
Fortran 2	008 binding		3	
MPI_Inter	comm_merge(intercomm, hig	ch, newintracomm, ierror)	3	
TYPE(	MPI_Comm), INTENT(IN) ::	intercomm	3	
LOGIC	LOGICAL, INTENT(IN) :: high			
TYPE(	MPI_Comm), INTENT(OUT) ::	newintracomm	3	
INTEG	ER, OPTIONAL, INTENT(OUT)	:: ierror	3	

Fortran binding MPI\_INTERCOMM\_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) INTEGER INTERCOMM, NEWINTRACOMM, IERROR LOGICAL HIGH

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the "low" group 





As shown in Figure 7.5, groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

int main(int argc, char \*argv[])

45

 $46 \\ 47$ 

```
1
        {
\mathbf{2}
          MPI_Comm
                      myComm;
                                    /* intra-communicator of local sub-group */
3
          MPI_Comm
                      myFirstComm; /* inter-communicators */
4
          MPI_Comm
                      mySecondComm;
5
          int membershipKey;
6
          int rank;
7
8
          MPI_Init(&argc, &argv);
9
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
          . . .
11
12
          /* User code must generate membershipKey in the range [0, 1, 2] */
13
          membershipKey = rank % 3;
14
15
          /* Build intra-communicator for local sub-group */
16
          MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
17
18
          /* Build inter-communicators. Tags are hard-coded. */
19
          if (membershipKey == 0)
20
          ſ
                         /* Group 0 communicates with groups 1 and 2. */
21
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
22
                                   1, &myFirstComm);
23
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
24
                                   2, &mySecondComm);
25
          }
26
          else if (membershipKey == 1)
27
          {
                     /* Group 1 communicates with groups 0 and 2. */
28
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
29
                                   1, &myFirstComm);
30
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2,
31
                                   12, &mySecondComm);
32
          }
33
          else if (membershipKey == 2)
34
                    /* Group 2 communicates with groups 0 and 1. */
          {
35
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
36
                                   2, &myFirstComm);
37
            MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
38
                                   12, &mySecondComm);
39
          }
40
41
          /* Do some work ... */
42
43
          /* Then free communicators before terminating... */
44
          MPI_Comm_free(&myFirstComm);
45
          MPI_Comm_free(&mySecondComm);
          MPI_Comm_free(&myComm);
46
47
          MPI_Finalize();
48
          return 0;
```

}

# 7.7 Caching

MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects: communicators, windows, and datatypes. More precisely, the caching facility allows a portable library to do the following:

- pass information between calls by associating it with an MPI intra- or inter-communicator, window, or datatype,
- quickly retrieve that information, and
- be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI\_COMM\_SELF is a suitable choice for posting process-local attributes, via this attribute-caching mechanism. (*End of advice to* users.)

*Rationale.* In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

One difficulty is the potential for size differences between Fortran integers and C pointers. For this reason, the Fortran versions of these routines use integers of kind MPI\_ADDRESS\_KIND.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI\_XXX\_CREATE\_KEYVAL is used with an object of the wrong type with a call to MPI\_YYY\_GET\_ATTR, MPI\_YYY\_SET\_ATTR, MPI\_YYY\_DELETE\_ATTR, or MPI\_YYY\_FREE\_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

## 7.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local <sup>46</sup> to the process and specific to the communicator to which they are attached. Attributes are <sup>47</sup> not propagated by MPI from one communicator to another except when the communicator is <sup>48</sup>

 $^{24}$ 

 $^{31}$ 

1 2 3	duplicated using MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO, and MPI_COMM_IDUP_WITH_INFO (and even then the application must give specific permission through callback functions for the attribute to be copied).
4	· · · · · · · · · · · · · · · ·
5	Advice to users. Attributes in C are of type void*. Typically, such an attribute will
6	be a pointer to a structure that contains further information, or a handle to an MPI
7	object. In Fortran, attributes are of type INTEGER. Such attribute can be a handle to
8	an MPI object, or just an integer-valued attribute. ( <i>End of advice to users.</i> )
9	an wir object, of just an integer-valued attribute. ( <i>Litu of autoice to users.</i> )
10	Advice to implementors. Attributes are scalar values, equal in size to, or larger than
	a C-language pointer. Attributes can always hold an MPI handle. ( <i>End of advice to</i>
11	implementors.)
12	implementors.)
13	The caching interface defined here requires that attributes be stored by MPI opaquely
14 15	within a communicator, window, or datatype. Accessor functions include the following:
16	• obtain a key value (used to identify an attribute); the user specifies "callback" func-
17	tions by which MPI informs the application when the communicator is destroyed or
18	copied.
19	copical and a second seco
20	• store and retrieve the value of an attribute;
21	
22	Advice to implementors. Caching and callback functions are only called synchronously,
23	in response to explicit application requests. This avoids problems that result from re-
24	peated crossings between user and system space. (This synchronous calling rule is a
25	general property of MPI.)
26	The choice of key values is under control of MPI. This allows MPI to optimize its
27	
28	implementation of attribute sets. It also avoids conflict between independent modules
29	caching information on the same communicators.
30	A much smaller interface, consisting of just a callback facility, would allow the entire
31	caching facility to be implemented by portable code. However, with the minimal call-
32	back interface, some form of table searching is implied by the need to handle arbitrary
	communicators. In contrast, the more complete interface defined here permits rapid
33	access to attributes through the use of pointers in communicators (to find the attribute
34	table) and cleverly chosen key values (to retrieve individual attributes). In light of the
35	efficiency "hit" inherent in the minimal interface, the more complete interface defined
36	here is seen to be superior. ( <i>End of advice to implementors.</i> )
37	here is seen to be superior. (End of dublee to implementors.)
38	MPI provides the following services related to caching. They are all process local.
39	with provides the following services related to eaching. They are an process local.
40	
41	7.7.2 Communicators
42 43	Functions for caching on communicators are:
44	
44	
45	
40	
48	
10	

MPI_CO	MM_CREATE_KEYVAL(comn extra_state)	n_copy_attr_fn, comm_delete_attr_fn, comm_keyval,	1 $2$
IN	comm_copy_attr_fn	copy callback function for comm_keyval (function)	3
IN	comm_delete_attr_fn	delete callback function for comm_keyval (function)	4 5
OUT	comm_keyval	key value for future access (integer)	6
IN	extra_state	extra state for callback function	7 8
C bindi int MPI	_Comm_create_keyval(MPI_(	<pre>Comm_copy_attr_function *comm_copy_attr_fn, tr_function *comm_delete_attr_fn, void *extra_state)</pre>	9 10 11 12 13
Fortran	2008 binding		14
MPI_Com	<pre>m_create_keyval(comm_copy extra_state, ierro</pre>	<pre>v_attr_fn, comm_delete_attr_fn, comm_keyval, r)</pre>	15 16
PRO	CEDURE(MPI_Comm_copy_att)	_function) :: comm_copy_attr_fn	17
		<pre>tr_function) :: comm_delete_attr_fn</pre>	18
	EGER, INTENT(OUT) :: comm	0	19 20
	EGER(KIND=MP1_ADDRESS_KIN EGER, OPTIONAL, INTENT(OU	ND), INTENT(IN) :: extra_state	21
			22
MPI_COM EXT INT	EXTRA_STATE, IERRO ERNAL COMM_COPY_ATTR_FN, EGER COMM_KEYVAL, IERROR	COMM_DELETE_ATTR_FN	23 24 25 26 27
TN.L	EGER(KIND=MPI_ADDRESS_KIN	ID) EXTRA_STATE	28
user, the used to a The C ca	bugh they are explicitly store associate attributes and acces allback functions are: int MPI_Comm_copy_attr_1	Keys are locally unique in a process, and opaque to d in integers. Once allocated, the key value can be s them on any locally defined communicator. Eunction(MPI_Comm oldcomm, int comm_keyval, void *attribute_val_in, l_out, int *flag);	29 30 31 32 33 34 35 36
and			37
typedef		<pre>c_function(MPI_Comm comm, int comm_keyval, l, void *extra_state);</pre>	38 39
With the ABSTRAC SUBRO	e mpi_f08 module, the Fortra I INTERFACE UTINE MPI_Comm_copy_attr_	function(oldcomm, comm_keyval, extra_state, attribute_val_out, flag, ierror)	40 41 42 43 44 45 46
	,		47 48
			10

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
\mathbf{2}
                     attribute_val_out
3
         LOGICAL :: flag
4
     and
5
     ABSTRACT INTERFACE
6
       SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
7
                     attribute_val, extra_state, ierror)
8
         TYPE(MPI_Comm) :: comm
9
         INTEGER :: comm_keyval, ierror
10
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
11
12
     With the mpi module and mpif.h, the Fortran callback functions are:
13
     SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
14
                    ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
15
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
16
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
17
                     ATTRIBUTE_VAL_OUT
18
         LOGICAL FLAG
19
     and
20
     SUBROUTINE COMM_DELETE_ATTR_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
21
                    EXTRA_STATE, IERROR)
22
         INTEGER COMM, COMM_KEYVAL, IERROR
23
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
^{24}
25
         The comm_copy_attr_fn function is invoked when a communicator is duplicated by
26
     MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO or
27
     MPI_COMM_IDUP_WITH_INFO. comm_copy_attr_fn should be of type
^{28}
     MPI_Comm_copy_attr_function. The copy callback function is invoked for each key value in
29
     oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its
30
     corresponding attribute. If it returns flag = 0 or .FALSE., then the attribute is deleted in
^{31}
     the duplicated communicator. Otherwise (flag = 1 or .TRUE.), the new attribute value is
32
     set to the value returned in attribute_val_out. The function returns MPI_SUCCESS on success
33
     and an error code on failure (in which case MPI_COMM_DUP or MPI_COMM_IDUP will
34
     fail).
35
         The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN
36
     or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a
37
     function that does nothing other than returning flag = 0 or .FALSE. (depending on whether
38
     the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and
39
     MPI_SUCCESS. MPI_COMM_DUP_FN is a simple copy function that sets flag = 1 or .TRUE.,
40
     returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. These
^{41}
     replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose
42
     use is deprecated.
43
44
                            Even though both formal arguments attribute_val_in and
           Advice to users.
45
           attribute_val_out are of type void*, their usage differs. The C copy function is passed
46
          by MPI in attribute_val_in the value of the attribute, and in attribute_val_out the
47
           address of the attribute, so as to allow the function to return the (new) attribute
48
           value. The use of type void* for both is to avoid messy type casts.
```

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to oldcomm only). (*End of advice to users.*)

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (*End of advice to implementors.*)

Analogous to comm\_copy\_attr\_fn is a callback deletion function, defined as follows. The comm\_delete\_attr\_fn function is invoked when a communicator is deleted by MPI\_COMM\_FREE or when a call is made explicitly to MPI\_COMM\_DELETE\_ATTR. comm\_delete\_attr\_fn should be of type MPI\_Comm\_delete\_attr\_function.

This function is called by MPI\_COMM\_FREE, MPI\_COMM\_DELETE\_ATTR, and MPI\_COMM\_SET\_ATTR to do whatever is needed to remove an attribute. The function returns MPI\_SUCCESS on success and an error code on failure (in which case MPI\_COMM\_FREE will fail).

The argument comm\_delete\_attr\_fn may be specified as

MPI\_COMM\_NULL\_DELETE\_FN from either C or Fortran.

MPI\_COMM\_NULL\_DELETE\_FN is a function that does nothing, other than returning MPI\_SUCCESS. MPI\_COMM\_NULL\_DELETE\_FN replaces MPI\_NULL\_DELETE\_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI\_SUCCESS, then the call that caused it to be invoked (for example, MPI\_COMM\_FREE), is erroneous.

The special key value MPI\_KEYVAL\_INVALID is never returned by MPI\_COMM\_CREATE\_KEYVAL. Therefore, it can be used for static initialization of key values.

*Advice to implementors.* The predefined Fortran functions MPI\_COMM\_NULL\_COPY\_FN, MPI\_COMM\_DUP\_FN, and

MPI\_COMM\_NULL\_DELETE\_FN are defined in the mpi module (and mpif.h) and the mpi\_f08 module with the same name, but with different interfaces. Each function can coexist twice with the same name in the same MPI library, one routine as an implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other routine within mpi\_f08 declared with CONTAINS. These routines have different link names, which are also different to the link names used for the routines used in C. (End of advice to implementors.)

Advice to users. Callbacks, including the predefined Fortran functions MPI\_COMM\_NULL\_COPY\_FN, MPI\_COMM\_DUP\_FN, and MPI\_COMM\_NULL\_DELETE\_FN should not be passed from one application routine that uses the mpi\_f08 module to another application routine that uses the mpi module or mpif.h, and vice versa; see also the advice to users on page 848. (End of advice to users.) 1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29 30

 $^{31}$ 

32

33

34

35

36

37

38

39 40

41

42

43

44

45

```
1
     MPI_COMM_FREE_KEYVAL(comm_keyval)
\mathbf{2}
       INOUT
                 comm_keyval
                                             key value (integer)
3
4
     C binding
5
     int MPI_Comm_free_keyval(int *comm_keyval)
6
\overline{7}
     Fortran 2008 binding
8
     MPI_Comm_free_keyval(comm_keyval, ierror)
9
          INTEGER, INTENT(INOUT) :: comm_keyval
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     Fortran binding
12
     MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
13
          INTEGER COMM_KEYVAL, IERROR
14
15
         Frees an extant attribute key. This function sets the value of keyval to
16
     MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use,
17
     because the actual free does not transpire until after all references (in other communicators
18
     on the process) to the key have been freed. These references need to be explicitly freed by the
19
     program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance,
20
     or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed
21
     communicator.
22
23
     MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)
^{24}
25
       INOUT
                                             communicator to which attribute will be attached
                 comm
26
                                             (handle)
27
       IN
                 comm_keyval
                                             key value (integer)
28
       IN
                 attribute_val
                                             attribute value
29
30
31
     C binding
32
     int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
33
     Fortran 2008 binding
34
     MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
35
          TYPE(MPI_Comm), INTENT(IN) :: comm
36
          INTEGER, INTENT(IN) :: comm_keyval
37
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
38
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
^{41}
     MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
42
          INTEGER COMM, COMM_KEYVAL, IERROR
43
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
44
         This function stores the stipulated attribute value attribute_val for subsequent retrieval
45
     by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if
46
     MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback
47
     function comm_delete_attr_fn was executed), and a new value was next stored. The call
```

is erroneous if there is no key with value keyval; in particular MPI\_KEYVAL\_INVALID is an erroneous key value. The call will fail if the comm\_delete\_attr\_fn function returned an error code other than MPI\_SUCCESS.

				4
				5
MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)				
IN	J	comm	communicator to which the attribute is attached	7
			(handle)	8
IN	J	comm_keyval	key value (integer)	9
		attribute_val	• ( )	10
0	01	attribute_vai	attribute value, unless $flag = false$	11
0	UT	flag	false if no attribute is associated with the key	12
			(logical)	13
				14
Ch	oinding			15
int	MPI_Con	mm_get_attr(MPI_Comm com	m, int comm_keyval, void *attribute_val,	16
		int *flag)		17
For	tran 20	08 binding		18 19
		0	l, attribute_val, flag, ierror)	20
	•	PI_Comm), INTENT(IN) ::	0	20 21
	INTEGER, INTENT(IN) :: comm_keyval			21
		• • • • =	, INTENT(OUT) :: attribute_val	23
		L, INTENT(OUT) :: flag	,,,,,,,,	24
		R, OPTIONAL, INTENT(OUT)	:: ierror	25
-				26
	tran bir	0		27
MPI			L, ATTRIBUTE_VAL, FLAG, IERROR)	28
		R COMM, COMM_KEYVAL, IER		29
		R(KIND=MPI_ADDRESS_KIND)	AIIKIBUIE_VAL	30
	LOGICA	L FLAG		31
	Retrieve	es attribute value by key. T	The call is erroneous if there is no key with value	32

Retrieves attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on comm for that key; in such case, the call returns flag = false. In particular MPI\_KEYVAL\_INVALID is an erroneous key value.

Advice to users. The call to MPI\_Comm\_set\_attr passes in attribute\_val the value of the attribute; the call to MPI\_Comm\_get\_attr passes in attribute\_val the address of the location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void\*, then the actual attribute\_val parameter to MPI\_Comm\_set\_attr will be of type void\* and the actual attribute\_val parameter to MPI\_Comm\_get\_attr will be of type void\*\*. (End of advice to users.)

*Rationale.* The use of a formal parameter attribute\_val of type void\* (rather than void\*\*) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void\*. (*End of rationale.*)

 $\mathbf{2}$ 

2

3

4

5 6

 $\overline{7}$ 

8 9

10

11

12

13

14 15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27 28

29 30

31 32

33 34

35

36

37

38 39

40 41

42

43

44

 $45 \\ 46$ 

MPI\_COMM\_DELETE\_ATTR(comm, comm\_keyval) INOUT communicator from which the attribute is deleted comm (handle) IN comm\_keyval key value (integer) C binding int MPI\_Comm\_delete\_attr(MPI\_Comm comm, int comm\_keyval) Fortran 2008 binding MPI\_Comm\_delete\_attr(comm, comm\_keyval, ierror) TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: comm\_keyval INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI\_COMM\_DELETE\_ATTR(COMM, COMM\_KEYVAL, IERROR) INTEGER COMM, COMM\_KEYVAL, IERROR Delete attribute from cache by key. This function invokes the attribute delete function comm\_delete\_attr\_fn specified when the keyval was created. The call will fail if the comm\_delete\_attr\_fn function returns an error code other than MPI\_SUCCESS. Whenever a communicator is replicated using the function MPI\_COMM\_DUP, MPI\_COMM\_IDUP, MPI\_COMM\_DUP\_WITH\_INFO or MPI\_COMM\_IDUP\_WITH\_INFO, all call-back copy functions for attributes that are currently set are invoked (in arbitrary order). Whenever a communicator is deleted using the function MPI\_COMM\_FREE all callback delete functions for attributes that are currently set are invoked. 7.7.3 Windows The functions for caching on windows are: MPI\_WIN\_CREATE\_KEYVAL(win\_copy\_attr\_fn, win\_delete\_attr\_fn, win\_keyval, extra\_state) IN win\_copy\_attr\_fn copy callback function for win\_keyval (function) win\_delete\_attr\_fn delete callback function for win\_keyval (function) IN OUT win\_keyval key value for future access (integer) IN extra\_state extra state for callback function C binding int MPI\_Win\_create\_keyval(MPI\_Win\_copy\_attr\_function \*win\_copy\_attr\_fn, MPI\_Win\_delete\_attr\_function \*win\_delete\_attr\_fn, int \*win\_keyval, void \*extra\_state) Fortran 2008 binding

<sup>47</sup> MPI\_Win\_create\_keyval(win\_copy\_attr\_fn, win\_delete\_attr\_fn, win\_keyval, <sup>48</sup> extra\_state, ierror)

PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn	1
PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn	2
INTEGER, INTENT(OUT) :: win_keyval	3
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5
Fortner hinding	6
Fortran binding	7
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	8
EXTRA_STATE, IERROR)	9
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	10
INTEGER WIN_KEYVAL, IERROR	11
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	12
The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or	13
MPI_WIN_DUP_FN from either C or Fortran. MPI_WIN_NULL_COPY_FN is a function	14
that does nothing other than returning $flag = 0$ and MPI_SUCCESS. MPI_WIN_DUP_FN is	15
a simple copy function that sets $flag = 1$ , returns the value of attribute_val_in in	16
attribute_val_out, and returns MPI_SUCCESS.	17
The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN	18
from either C or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing,	19
other than returning MPI_SUCCESS.	20
The C callback functions are:	21
<pre>typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,</pre>	22
void *extra_state, void *attribute_val_in,	23
<pre>void *attribute_val_out, int *flag);</pre>	24
,	25
and	26
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,	27
<pre>void *attribute_val, void *extra_state);</pre>	28
With the mpi_f08 module, the Fortran callback functions are:	29
ABSTRACT INTERFACE	30
SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,	31
attribute_val_in, attribute_val_out, flag, ierror)	32
TYPE(MPI_Win) :: oldwin	33
INTEGER :: win_keyval, ierror	34
INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,	35
attribute_val_out	36
LOGICAL :: flag	37
	38
and	39
ABSTRACT INTERFACE	40
SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,	41
extra_state, ierror)	42
TYPE(MPI_Win) :: win	43
INTEGER :: win_keyval, ierror	44
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state</pre>	45
With the mpi module and mpif.h, the Fortran callback functions are:	46
- · · ·	47

```
1
     SUBROUTINE WIN_COPY_ATTR_FUNCTION(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
\mathbf{2}
                    ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
3
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
4
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
5
                     ATTRIBUTE_VAL_OUT
6
         LOGICAL FLAG
7
     and
8
     SUBROUTINE WIN_DELETE_ATTR_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
9
                    EXTRA_STATE, IERROR)
10
         INTEGER WIN, WIN_KEYVAL, IERROR
11
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
12
13
         If an attribute copy function or attribute delete function returns other than
14
     MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is
15
     erroneous.
16
17
     MPI_WIN_FREE_KEYVAL(win_keyval)
18
19
       INOUT
                win_keyval
                                            key value (integer)
20
21
     C binding
22
     int MPI_Win_free_keyval(int *win_keyval)
23
     Fortran 2008 binding
^{24}
     MPI_Win_free_keyval(win_keyval, ierror)
25
          INTEGER, INTENT(INOUT) :: win_keyval
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     Fortran binding
29
     MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
30
         INTEGER WIN_KEYVAL, IERROR
^{31}
32
33
     MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
34
       INOUT
35
                win
                                            window to which attribute will be attached (handle)
36
       IN
                win_keyval
                                           key value (integer)
37
       IN
                attribute_val
                                           attribute value
38
39
     C binding
40
     int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)
41
42
     Fortran 2008 binding
43
     MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
44
         TYPE(MPI_Win), INTENT(IN) :: win
45
         INTEGER, INTENT(IN) :: win_keyval
46
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

Fortran binding MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR) INTEGER WIN, WIN_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL					
MPI WIN	_GET_ATTR(win, win_keyval,	attribute val flag)	6 7		
IN	win	•,	8		
		window to which the attribute is attached (handle)	9		
IN	win_keyval	key value (integer)	10 11		
OUT	attribute_val	attribute value, unless $flag = false$	11		
OUT	flag	false if no attribute is associated with the key (logical)	13 14		
			15		
C binding			16		
int MPI_W	0	<pre>int win_keyval, void *attribute_val,</pre>	17		
	int *flag)		18 19		
	2008 binding		20		
	· ·	attribute_val, flag, ierror)	21		
	MPI_Win), INTENT(IN) :: ER, INTENT(IN) :: win_ke		22		
		), INTENT(OUT) :: attribute_val	23		
	CAL, INTENT(OUT) :: flag	,,	24		
INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding					
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)					
INTEGER WIN, WIN_KEYVAL, IERROR					
INTEC	ER(KIND=MPI_ADDRESS_KIND	) ATTRIBUTE_VAL	30		
LOGIC	CAL FLAG		31		
			32		
		N	33		
MPI_WIN_	_DELETE_ATTR(win, win_key	val)	34 35		
INOUT	win	window from which the attribute is deleted (handle)	36		
IN	win_keyval	key value (integer)	37		
			38		
C binding	g		39		
int MPI_W	<pre>int MPI_Win_delete_attr(MPI_Win win, int win_keyval)</pre>				
Fortran 2008 binding					
MPI_Win_d	lelete_attr(win, win_keyv	al, ierror)	42 43		
	TYPE(MPI_Win), INTENT(IN) :: win				
	INTEGER, INTENT(IN) :: win_keyval				
INTEC	ER, OPTIONAL, INTENT(OUT	) :: lerror	46		
Fortran binding					
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR) 48					

```
1
         INTEGER WIN, WIN_KEYVAL, IERROR
\mathbf{2}
3
     7.7.4 Datatypes
4
\mathbf{5}
     The new functions for caching on datatypes are:
6
7
     MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
8
9
                    extra_state)
10
       IN
                 type_copy_attr_fn
                                            copy callback function for type_keyval (function)
11
       IN
                 type_delete_attr_fn
                                            delete callback function for type_keyval (function)
12
       OUT
                type_keyval
                                            key value for future access (integer)
13
14
       IN
                extra_state
                                            extra state for callback function
15
16
     C binding
17
     int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,
18
                    MPI_Type_delete_attr_function *type_delete_attr_fn,
19
                    int *type_keyval, void *extra_state)
20
     Fortran 2008 binding
21
22
     MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
23
                    extra_state, ierror)
^{24}
         PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
25
         PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
26
         INTEGER, INTENT(OUT) :: type_keyval
27
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     Fortran binding
30
     MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
^{31}
                    EXTRA_STATE, IERROR)
32
         EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
33
         INTEGER TYPE_KEYVAL, IERROR
34
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
35
36
         The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or
37
     MPI_TYPE_DUP_FN from either C or Fortran. MPI_TYPE_NULL_COPY_FN is a function
38
     that does nothing other than returning flag = 0 and MPI_SUCCESS. MPI_TYPE_DUP_FN
39
     is a simple copy function that sets flag = 1, returns the value of attribute_val_in in
40
     attribute_val_out, and returns MPI_SUCCESS.
41
         The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN
42
     from either C or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing,
43
     other than returning MPI_SUCCESS.
44
     The C callback functions are:
45
     typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
46
                    int type_keyval, void *extra_state, void *attribute_val_in,
47
                    void *attribute_val_out, int *flag);
48
```

```
1
and
                                                                                     2
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
                                                                                     3
              int type_keyval, void *attribute_val, void *extra_state);
                                                                                     4
With the mpi_f08 module, the Fortran callback functions are:
                                                                                      5
ABSTRACT INTERFACE
                                                                                      6
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                     7
               attribute_val_in, attribute_val_out, flag, ierror)
                                                                                      8
    TYPE(MPI_Datatype) :: oldtype
                                                                                     9
    INTEGER :: type_keyval, ierror
                                                                                     10
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                     11
               attribute_val_out
                                                                                     12
    LOGICAL :: flag
                                                                                     13
                                                                                     14
and
                                                                                     15
ABSTRACT INTERFACE
                                                                                     16
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                     17
               attribute_val, extra_state, ierror)
                                                                                     18
    TYPE(MPI_Datatype) :: datatype
                                                                                     19
    INTEGER :: type_keyval, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                     20
                                                                                     21
With the mpi module and mpif.h, the Fortran callback functions are:
                                                                                     22
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
                                                                                     23
              ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
                                                                                     24
    INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
                                                                                     25
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                                                                                     26
               ATTRIBUTE_VAL_OUT
                                                                                     27
    LOGICAL FLAG
                                                                                     28
                                                                                     29
and
                                                                                     30
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
                                                                                     31
              EXTRA_STATE, IERROR)
                                                                                     32
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
                                                                                     33
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
                                                                                     34
    If an attribute copy function or attribute delete function returns other than
                                                                                     35
MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),
                                                                                     36
is erroneous.
                                                                                     37
                                                                                     38
                                                                                     39
MPI_TYPE_FREE_KEYVAL(type_keyval)
                                                                                     40
          type_keyval
 INOUT
                                     key value (integer)
                                                                                     41
                                                                                     42
C binding
                                                                                     43
int MPI_Type_free_keyval(int *type_keyval)
                                                                                     44
                                                                                     45
Fortran 2008 binding
                                                                                     46
MPI_Type_free_keyval(type_keyval, ierror)
                                                                                     47
    INTEGER, INTENT(INOUT) :: type_keyval
                                                                                     48
```

```
1
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     Fortran binding
3
     MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
4
          INTEGER TYPE_KEYVAL, IERROR
5
6
7
     MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)
8
9
       INOUT
                 datatype
                                             datatype to which attribute will be attached (handle)
10
       IN
                 type_keyval
                                            key value (integer)
11
       IN
                 attribute_val
                                            attribute value
12
13
14
     C binding
15
     int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
16
                    void *attribute_val)
17
     Fortran 2008 binding
18
     MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
19
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
          INTEGER, INTENT(IN) :: type_keyval
21
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
22
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
^{24}
     Fortran binding
25
     MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
26
          INTEGER DATATYPE, TYPE_KEYVAL, IERROR
27
          INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
28
29
30
     MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)
^{31}
       IN
                 datatype
                                             datatype to which the attribute is attached (handle)
32
33
       IN
                 type_keyval
                                            key value (integer)
34
       OUT
                 attribute_val
                                            attribute value, unless flag = false
35
       OUT
                 flag
                                             false if no attribute is associated with the key
36
                                             (logical)
37
38
     C binding
39
     int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
40
                    void *attribute_val, int *flag)
41
42
     Fortran 2008 binding
43
     MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
44
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
          INTEGER, INTENT(IN) :: type_keyval
46
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
47
          LOGICAL, INTENT(OUT) :: flag
48
```

Fortran h MPI_TYPE_ INTEC	0	KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR) , IERROR	1 2 3 4 5 6 7 8 9	
MPI_TYPI	E_DELETE_ATTR(datatype, ty	/pe_keyval)	10	
INOUT	datatype	data type from which the attribute is deleted (handle)	11 12	
IN	type_keyval	key value (integer)	12	
C bindin int MPI_7	0	type datatype, int type_keyval)	14 15 16	
Fortran 2	2008 binding		17	
	delete_attr(datatype, ty	· ·	18 19	
	(MPI_Datatype), INTENT(IN) EER, INTENT(IN) :: type_k		20	
	ER, OPTIONAL, INTENT(OUT)		21	
Fortran h	ainding		22	
Fortran binding MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)			23 24	
INTEGER DATATYPE, TYPE_KEYVAL, IERROR			25	
			26	
7.7.5 Error Class for Invalid Keyval			27 28	
Kev values	s for attributes are system-allo	ocated, by	29	
-	-	uch values can be passed to the functions that use	30	
÷	· 0	er to signal that an erroneous key value has been	31	
·	,	s a new MPI error class: MPI_ERR_KEYVAL. It can _ATTR_GET, MPI_ATTR_DELETE,	32 33	
	VAL_FREE,	ATTR_GET, MFT_ATTR_DELETE,	34	
	<pre>{}_DELETE_ATTR,</pre>		35	
	<pre>{}_SET_ATTR,</pre>		36	
	<pre>{}_GET_ATTR, </pre>		37	
C C	M_DUP_WITH_INFO, MPI_CON	IM_DUP, MPI_COMM_IDUP,	38 39	
—	, _	COMM_FREE. The last six are included because	40	
keyval is a	n argument to the copy and d	elete functions for attributes.	41	
7.7.6 At	tributes Example		42 43 44	
Advi	ce to users. This example	e shows how to write a collective communication	44 45	
operation that uses caching to be more efficient after the first call. ( <i>End of advice to</i> $_{46}$ $_{47}$				

```
1
        /* key for this module's stuff: */
\mathbf{2}
        static int gop_key = MPI_KEYVAL_INVALID;
3
4
        typedef struct
5
        ſ
6
           int ref_count;
                                     /* reference count */
7
           /* other stuff, whatever else we want */
8
        } gop_stuff_type;
9
10
        void Efficient_Collective_Op(MPI_Comm comm, ...)
11
        {
12
          gop_stuff_type *gop_stuff;
13
          MPI_Group
                           group;
14
          int
                           foundflag;
15
16
          MPI_Comm_group(comm, &group);
17
18
          if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
19
          ſ
20
            if ( ! MPI_Comm_create_keyval(gop_stuff_copier,
21
                                       gop_stuff_destructor,
22
                                       &gop_key, NULL)) {
23
            /* get the key while assigning its copy and delete callback
24
                behavior. */
25
            } else
26
                MPI_Abort(comm, 99);
27
          }
28
29
          MPI_Comm_get_attr(comm, gop_key, &gop_stuff, &foundflag);
30
          if (foundflag)
31
          { /* This module has executed in this group before.
32
                We will use the cached information */
33
          }
34
          else
35
          { /* This is a group that we have not yet cached anything in.
36
                We will now do so.
37
            */
38
39
            /* First, allocate storage for the stuff we want,
40
                and initialize the reference count */
41
42
            gop_stuff = (gop_stuff_type *) malloc(sizeof(gop_stuff_type));
43
            if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
44
45
            gop_stuff->ref_count = 1;
46
47
            /* Second, fill in *gop_stuff with whatever we want.
48
                This part isn't shown here */
```

```
/* Third, store gop_stuff as the attribute value */
    MPI_Comm_set_attr(comm, gop_key, gop_stuff);
  }
  /* Then, in any case, use contents of *gop_stuff
     to do the global op ... */
}
/* The following routine is called by MPI when a group is freed */
int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
                         void *extra)
{
  gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff->ref_count -= 1;
  /* If no references remain, then free the storage */
  if (gop_stuff->ref_count == 0) {
    free((void *)gop_stuff);
  }
  return MPI_SUCCESS;
}
/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
               void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
{
  gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
  gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
  if (keyval != gop_key) { /* abort -- programming error */ }
  /* The new group adds one reference to this gop_stuff */
  gop_stuff_in->ref_count += 1;
  *gop_stuff_out = gop_stuff_in;
  return MPI_SUCCESS;
}
```

# 7.8 Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

1 2

3

4

5 6

7

9 10 11

12

13

 $14 \\ 15$ 

16 17

18

19

20 21

22

23 24

25

26

27

28 29

30

31

32

33

34

35 36

37

38

39 40 41

42 43

44

45

46

47

MPI\_COMM\_SET\_NAME(comm, comm\_name)

```
2
       INOUT
                                              communicator whose identifier is to be set (handle)
                 comm
3
       IN
                                              the character string which is remembered as the
                 comm_name
4
                                              name (string)
5
6
7
     C binding
8
     int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
9
     Fortran 2008 binding
10
     MPI_Comm_set_name(comm, comm_name, ierror)
11
          TYPE(MPI_Comm), INTENT(IN) :: comm
12
          CHARACTER(LEN=*), INTENT(IN) :: comm_name
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
17
          INTEGER COMM, IERROR
18
          CHARACTER*(*) COMM_NAME
19
          MPI_COMM_SET_NAME allows a user to associate a name string with a communicator.
20
     The character string which is passed to MPI_COMM_SET_NAME will be saved inside the
21
     MPI library (so it can be freed by the caller immediately after the call, or allocated on the
22
     stack). Leading spaces in name are significant but trailing ones are not.
23
          MPI_COMM_SET_NAME is a local (non-collective) operation, which only affects the
^{24}
     name of the communicator as seen in the process which made the MPI_COMM_SET_NAME
25
     call. There is no requirement that the same (or any) name be assigned to a communicator
26
     in every process where it exists.
27
28
           Advice to users. Since MPI_COMM_SET_NAME is provided to help debug code, it
29
           is sensible to give the same name to a communicator in all of the processes where it
30
           exists, to avoid confusion. (End of advice to users.)
^{31}
32
          The length of the name which can be stored is limited to the value of
33
     MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C to allow for the
34
     null terminator. Attempts to put names longer than this will result in truncation of the
35
     name. MPI_MAX_OBJECT_NAME must have a value of at least 64.
36
37
           Advice to users. Under circumstances of store exhaustion an attempt to put a name
38
           of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be
39
           viewed only as a strict upper bound on the name length, not a guarantee that setting
40
           names of less than this length will always succeed. (End of advice to users.)
41
42
           Advice to implementors. Implementations which pre-allocate a fixed size space for a
43
           name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME.
44
           Implementations which allocate space for the name from the heap should still define
45
           MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate
46
           space for a string of up to this size when calling MPI_COMM_GET_NAME. (End of
47
           advice to implementors.)
48
```

MPI_COMM_GET_NAME(comm, comm_name, resultlen)				
IN	comm communicator whose name is to be returned (handle)			
OUT	comm_name	the name previously stored on the communicator, or an empty string if no such name exists (string)		
OUT	resultlen	length of returned name (integer)		
C binding int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)				
<pre>Fortran 2008 binding MPI_Comm_get_name(comm, comm_name, resultlen, ierror) TYPE(MPI_Comm), INTENT(IN) :: comm CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name INTEGER, INTENT(OUT) :: resultlen INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>				
Fortran binding				

MPI_COMM_GET_NAME(	COMM, COMM	_NAME,	RESULTLEN,	IERROR)
INTEGER COMM,	RESULTLEN,	IERRO	ર	
CHARACTER*(*)	COMM NAME			

MPI\_COMM\_GET\_NAME returns the last name which has previously been associated with the given communicator. The name may be set and retrieved from any language. The same name will be returned independent of the language used. comm\_name should be allocated so that it can hold a resulting string of length MPI\_MAX\_OBJECT\_NAME characters. MPI\_COMM\_GET\_NAME returns a copy of the set name in comm\_name.

In C, a null character is additionally stored at comm\_name[resultlen]. The value of resultlen cannot be larger than MPI\_MAX\_OBJECT\_NAME-1. In Fortran, comm\_name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI\_MAX\_OBJECT\_NAME.

If the user has not associated a name with a communicator, or an error occurs, MPI\_COMM\_GET\_NAME will return an empty string (all spaces in Fortran, "" in C). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI\_COMM\_WORLD, MPI\_COMM\_SELF, and the communicator returned by MPI\_COMM\_GET\_PARENT (if not MPI\_COMM\_NULL) will have the default of "MPI\_COMM\_WORLD", "MPI\_COMM\_SELF", and "MPI\_COMM\_PARENT". The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

*Rationale.* We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.

 $^{24}$ 

1 2		ribute key useful additional code to call <b>strdup</b> is necessary. If			
3	which we can ea	ardized then users have to write it. This is extra unneeded work sily eliminate.			
4	• The Fortran hinding is not trivial to write (it will depend on details of the				
5	• The Fortran binding is not trivial to write (it will depend on details of the				
6	Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.				
7	the library rathe	er than in user code.			
8	(End of rationale.)				
9					
10	Advice to users. The	e above definition means that it is safe simply to print the string			
11	returned by MPI_CO	MM_GET_NAME, as it is always a valid string even if there was			
12	no name.				
13	Note that associating	a name with a communicator has no effect on the semantics of			
14	0				
15		will (necessarily) increase the store requirement of the program,			
16		be saved. Therefore there is no requirement that users use these			
17		e names with communicators. However debugging and profiling			
18		y be made easier if names are associated with communicators,			
		r profiler should then be able to present information in a less			
19	cryptic manner. (End	l of advice to users.)			
20					
21	The following function	ns are used for setting and getting names of datatypes. The			
22	constant MPI_MAX_OBJECT	$\Gamma_NAME$ also applies to these names.			
23					
24					
25	MPI_TYPE_SET_NAME(da	atatype, type_name)			
26	INOUT datatype	datatype whose identifier is to be set (handle)			
27	IN type_name	the character string which is remembered as the			
28	type_name	name (string)			
29		hame (string)			
30					
31	C binding				
32	<pre>int MPI_Type_set_name(N</pre>	<pre>/PI_Datatype datatype, const char *type_name)</pre>			
33	Fortran 2008 binding				
34	6	una tuna noma iarran)			
35		cype, type_name, ierror)			
36		, INTENT(IN) :: datatype			
		<pre>INTENT(IN) :: type_name</pre>			
37	INTEGER, OPTIONAL,	INTENT(OUT) :: ierror			
38	Fortran binding				
39	0				
40		TYPE, TYPE_NAME, IERROR)			
41	INTEGER DATATYPE,				
42	CHARACTER*(*) TYPE_	NAME			
43					
44					
45					
46					
47					
48					
40					

MPI_TYF	PE_GET_NAME(datatyp	e, type_name, resultlen)	1			
IN	datatype	datatype whose name is to be returned (handle)	2			
OUT	type_name	the name previously stored on the datatype, or an	3 4			
	,	empty string if no such name exists (string)	5			
OUT	resultlen	length of returned name (integer)	6			
			7			
C bindi	ng		8			
	.nt MPI_Type_get_name(MPI_Datatype datatype, char *type_name,					
	int *resultler		10			
Fortran	2008 binding		11 12			
	-	type_name, resultlen, ierror)	12			
	C(MPI_Datatype), INT		14			
	• •	BJECT_NAME), INTENT(OUT) :: type_name	15			
INTE	GER, INTENT(OUT) ::	resultlen	16			
INTE	GER, OPTIONAL, INTE	NT(OUT) :: ierror	17			
Fortran	binding		18			
		TYPE_NAME, RESULTLEN, IERROR)	19			
	GER DATATYPE, RESUL		20			
CHAF	ACTER*(*) TYPE_NAME		21			
Nam	ed predefined detetype	s have the default names of the datatype name. For exam-	22 23			
		name of "MPI_WCHAR".	20			
<b>_</b> /		used for setting and getting names of windows. The con-	25			
	0	also applies to these names.	26			
			27			
			28			
	I_SET_NAME(win, win_	·	29			
INOUT	win	window whose identifier is to be set (handle)	30			
IN	win_name	the character string which is remembered as the	31			
		name (string)	32 33			
			34			
C bindin	-		35			
int MPI_	Win_set_name(MPI_Wi	n win, const char *win_name)	36			
Fortran	2008 binding		37			
MPI_Win_	_set_name(win, win_n	ame, ierror)	38			
TYPE	TYPE(MPI_Win), INTENT(IN) :: win3CHARACTER(LEN=*), INTENT(IN) :: win_name4					
INTE	CGER, OPTIONAL, INTE	NT(OUT) :: ierror	41			
Fortran	binding		42 43			
MPI_WIN_	MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)					
	GER WIN, IERROR		44 45			
CHAF	CHARACTER*(*) WIN_NAME 46					

MPI\_WIN\_GET\_NAME(win, win\_name, resultlen)

```
2
       IN
                                            window whose name is to be returned (handle)
                win
3
       OUT
                win_name
                                            the name previously stored on the window, or an
4
                                            empty string if no such name exists (string)
5
6
       OUT
                resultlen
                                            length of returned name (integer)
7
8
     C binding
9
     int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
10
     Fortran 2008 binding
11
     MPI_Win_get_name(win, win_name, resultlen, ierror)
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
14
         INTEGER, INTENT(OUT) :: resultlen
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     Fortran binding
18
     MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
19
         INTEGER WIN, RESULTLEN, IERROR
20
         CHARACTER*(*) WIN_NAME
21
```

7.9 Formalizing the Loosely Synchronous Model

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

7.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that 30 communicator must be free of side effects throughout execution of the subprogram: there  $^{31}$ 32 should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work "safely." For libraries 33 34so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will 35 interfere. Since we permit good implementations to create new communicators without 36 synchronization (such as by preallocated contexts on communicators), this does not impose 37 a significant overhead. 38

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

42 43

22 23

24 25

26

27 28

29

1

7.9.2 Models of Execution

<sup>44</sup> <sup>45</sup> In the loosely synchronous model, transfer of control to a **parallel procedure** is effected by <sup>46</sup> having each executing process invoke the procedure. The invocation is a collective operation: <sup>47</sup> it is executed by all processes in the execution group, and invocations are similarly ordered <sup>48</sup> at all processes. However, the invocation need not be synchronized. We say that a parallel procedure is *active* in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

### Static Communicator Allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

### Dynamic Communicator Allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI\_COMM\_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI\_COMM\_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

- messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
- messages are always selected by source (no use is made of MPI\_ANY\_SOURCE).

### The General Case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user's responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, communicator creation is properly coordinated. 1

 $\mathbf{2}$ 

3

4 5

6 7

8

9

10

11

12

13

14

15

16

17 18

19 20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38 39

40 41

42 43

44

45

46

47

1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23 24				
24 25				
26				
27				
28				
29				
30				
31				
32				
33				
34				
35				
36				
37				
38				
39				
40 41				
41 42				
42 43				
43 44				
45				
46				
47				
48				

# Chapter 8

# **Process Topologies**

### 8.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to intercommunicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

 $^{24}$ 

As stated in Chapter 7, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the "virtual topology."

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal with machine-independent mapping and communication on virtual process topologies.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [49]. On the other hand, if there is no way for the user to specify the logical process arrangement as a "virtual topology," a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [12, 13].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

# 8.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a "missing link" in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping.

16Specifying the virtual topology in terms of a graph is sufficient for all applications. 17However, in many applications the graph structure is regular, and the detailed set-up of the 18 graph would be inconvenient for the user and might be less efficient at run time. A large frac-19tion of all parallel applications use process topologies like rings, two- or higher-dimensional 20grids, or tori. These structures are completely defined by the number of dimensions and 21the numbers of processes in each coordinate direction. Also, the mapping of grids and tori 22is generally an easier problem than that of general graphs. Thus, it is desirable to address 23these cases explicitly. 24

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a  $(2 \times 2)$  grid is as follows.

coord $(0,0)$ :	rank 0
coord $(0,1)$ :	rank 1
coord $(1,0)$ :	$\operatorname{rank} 2$
coord $(1,1)$ :	rank 3

## 8.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 7.

40 Information representing an MPI virtual topology may be added to a communicator at  $^{41}$ the time of its creation. If a communicator creation function adds information representing 42an MPI virtual topology to the output communicator it creates, then it either propagates 43the topology representation from the input communicator to the output communicator, or 44adds a new topology representation generated from the input parameters that describe a 45virtual topology. The description of every MPI communicator creation function explicitly 46states how topology information is handled. Communicator creation functions that create 47new topology representations are described in Section 8.5. 48

1

2

3

 $\frac{4}{5}$ 

6 7

8

9

10

11

12

13

14

15

25

26

27

35 36

37

38

### 8.4 Overview of the Functions

MPI supports three topology types: **Cartesian**, **graph**, and **distributed graph**. The function MPI\_CART\_CREATE can be used to create Cartesian topologies, the function MPI\_GRAPH\_CREATE can be used to create graph topologies, and the functions MPI\_DIST\_GRAPH\_CREATE\_ADJACENT and MPI\_DIST\_GRAPH\_CREATE can be used to create distributed graph topologies. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The above topology creation functions take as input an existing communicator comm\_old, which defines the set of processes on which the topology is to be mapped. For MPI\_GRAPH\_CREATE and MPI\_CART\_CREATE, all input arguments must have identical values on all processes of the group of comm\_old. When calling MPI\_GRAPH\_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI\_DIST\_GRAPH\_CREATE\_ADJACENT or MPI\_DIST\_GRAPH\_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm\_topol is created that carries the topological structure as cached information (see Chapter 7). In analogy to function MPI\_COMM\_CREATE, no cached information propagates from comm\_old to comm\_topol.

MPI\_CART\_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an *n*-dimensional hypercube is an *n*-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI\_DIMS\_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

MPI defines functions to query a communicator for topology information. The function 29 MPI\_TOPO\_TEST is used to query for the type of topology associated with a communicator. 30 Depending on the topology type, different information can be extracted. For a graph 31topology, the functions MPI\_GRAPHDIMS\_GET and MPI\_GRAPH\_GET retrieve the graph-32 topology information that is associated with the communicator. Additionally, the functions 33 MPI\_GRAPH\_NEIGHBORS\_COUNT and MPI\_GRAPH\_NEIGHBORS can be used to obtain 34 the neighbors of an arbitrary node in the graph. For a distributed graph topology, the 35 functions MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT and MPI\_DIST\_GRAPH\_NEIGHBORS 36 can be used to obtain the neighbors of the calling process. For a Cartesian topology, the 37 function MPI\_CARTDIM\_GET returns the number of dimensions and 38 39

MPI\_CART\_GET returns the numbers of MPI processes in each dimension and periodicity of the associated Cartesian topology. Additionally, the functions MPI\_CART\_RANK and MPI\_CART\_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI\_CART\_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI\_CART\_SUB can be used to extract a Cartesian subspace (analogous to MPI\_COMM\_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI\_GRAPH\_MAP and MPI\_CART\_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation 48

1 2

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

40

41

42

43

44

45

1 2	functions presented in Chapter 7, they are sufficient to implement all other topology func-				
3	tions. Section 8.5.8 outlines such an implementation. The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER,				
4	MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL,				
5			and MPI_NEIGHBOR_ALLTOALLW communicate with the		
6		,	ogy associated with the communicator. The nonblocking		
7		9	ALLGATHER, MPI_INEIGHBOR_ALLGATHERV,		
8	MPI_INE	EIGHBOR_ALLTOALL,	MPI_INEIGHBOR_ALLTOALLV, and		
9	MPI_INE	EIGHBOR_ALLTOALLW			
10					
11 12	8.5 T	opology Constructo	ors		
13 14	8.5.1 (	Cartesian Constructor			
15					
16 17	MPI_CA	RT_CREATE(comm_old	, ndims, dims, periods, reorder, comm_cart)		
18	IN	comm_old	input communicator (handle)		
19	IN	ndims	number of dimensions of Cartesian grid (integer)		
20 21	IN	dims	integer array of size ndims specifying the number of		
22			processes in each dimension		
23 24	IN	periods	logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension		
25 26	IN	reorder	ranking may be reordered (true) or not (false) (logical)		
27 28	OUT	comm_cart	communicator with new Cartesian topology (handle)		
29	C bindi	ing			
30 31		0	nm comm_old, int ndims, const int dims[],		
32	1110 111 1		iods[], int reorder, MPI_Comm *comm_cart)		
33	<b>T</b> (	-			
34		2008 binding			
35	MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)				
36	TYPE(MPI_Comm), INTENT(IN) :: comm_old				
37	INTEGER, INTENT(IN) :: ndims, dims(ndims) LOGICAL, INTENT(IN) :: periods(ndims), reorder				
38	TYPE(MPI_Comm), INTENT(OUT) :: comm_cart				
39		EGER, OPTIONAL, INT			
40					
41 42		ו binding ד כפהאדה (כמאא מוס ו	NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)		
42			S, DIMS, PERIODS, REORDER, COMM_CARI, IERROR		
44		ICAL PERIODS(*), REG			
45		-			
46			ns a handle to a new communicator to which the Cartesian		
47	topology	information is attache	d. If reorder = false then the rank of each process in the		

new group is identical to its rank in the old group. Otherwise, the function may reorder 48

the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of comm\_old, then some processes are returned MPI\_COMM\_NULL, in analogy to MPI\_COMM\_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative. MPI\_CART\_CREATE will associate information representing a Cartesian topology with the specified number of dimensions, numbers of MPI processes in each coordinate direction, and periodicity with the new communicator.

## 8.5.2 Cartesian Convenience Function: MPI\_DIMS\_CREATE

For Cartesian topologies, the function MPI\_DIMS\_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI\_COMM\_WORLD's group) into an *n*-dimensional topology.

MPI_DIMS_CREATE(nnodes, ndims, dims)						
IN	nnodes	number of nodes in a grid (integer)				
IN	ndims	number of Cartesian dimensions (integer)				
INOUT	dims	integer array of size ndims specifying the number of nodes in each dimension				

### C binding

int MPI\_Dims\_create(int nnodes, int ndims, int dims[])

### Fortran 2008 binding

MPI\_Dims\_create(nnodes, ndims, dims, ierror)
 INTEGER, INTENT(IN) :: nnodes, ndims
 INTEGER, INTENT(INOUT) :: dims(ndims)
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

### Fortran binding

MPI\_DIMS\_CREATE(NNODES, NDIMS, DIMS, IERROR)
INTEGER NNODES, NDIMS, DIMS(\*), IERROR

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array dims. If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension i; only those entries where dims[i] = 0 are modified by the call.

Negative input values of  $\mathsf{dims}[i]$  are erroneous. An error will occur if  $\mathsf{nnodes}$  is not a multiple of

$$\prod_{i \in [i]} dims[i].$$

 $i, \mathsf{dims}[i] \neq 0$ 

 $46 \\ 47$ 

For dims[i] set by the call, dims[i] will be ordered in nonincreasing order. Array dims is suitable for use as input to routine MPI\_CART\_CREATE. MPI\_DIMS\_CREATE is local. If ndims is zero and nnodes is one, MPI\_DIMS\_CREATE returns MPI\_SUCCESS.

<b></b> pro	8.1						
	dims	function call		dims			
	before call			on return			
	(0,0)	MPI_DIMS_CREATE(6	, ,	(3,2)			
	(0,0)	MPI_DIMS_CREATE(7	,	(7,1)			
	$(0,3,0) \\ (0,3,0)$	MPI_DIMS_CREATE(6 MPI_DIMS_CREATE(7	,	(2,3,1) erroneous call			
	(0,5,0)		, <u>s</u> , unis)	cironeous can			
8.5.3 Gra	aph Constructor						
MPI_GRAF	PH_CREATE(con	nm_old, nnodes, index, e	lges, reorde	r, comm_graph)			
IN	comm_old	input com	municator (	(handle)			
IN	nnodes	number of	f nodes in gi	caph (integer)			
IN	index	array of in	ntegers descr	ribing node degrees	s (see belo		
IN	edges	array of in	array of integers describing graph edges (see below)				
IN	reorder	ranking m (logical)	ay be reord	ered (true) or not	(false)		
OUT	comm_graph	communio	ator with g	raph topology adde	ed (handle		
C binding int MPI_G	raph_create(M	PI_Comm comm_old, in edges[], int reorde					
Fortran 2	008 binding						
MPI_Graph	_create(comm_	old, nnodes, index,	edges, re	order, comm_gra	aph,		
	ierror)						
		TENT(IN) :: comm_old ) :: nnodes, index(n:	nodoc) -	daog (*)			
	AL, INTENT(IN)		noues), e	TRAP (*)			
		TENT(OUT) :: comm_gr	aph				
INTEGER, OPTIONAL, INTENT(OUT) :: ierror							
INTEG	Fortran binding						
Fortran b	oinding	DLD, NNODES, INDEX.	EDGES. RE	ORDER, COMM_GR	APH,		
Fortran b	oinding	DLD, NNODES, INDEX,	EDGES, RE	DRDER, COMM_GR	APH,		
Fortran b MPI_GRAPH	inding CREATE(COMM_( IERROR)	DLD, NNODES, INDEX, MODES, INDEX, MODES, INDEX(*), ED					
Fortran b MPI_GRAPH INTEG	inding CREATE(COMM_( IERROR)						
Fortran b MPI_GRAPH INTEG LOGIC	oinding _CREATE(COMM_( IERROR) ER COMM_OLD, 1 AL REORDER		GES(*), C	DMM_GRAPH, IER	ROR		

 $\mathbf{1}$ 

 $\mathbf{2}$ 

processes. If the size, nnodes, of the graph is smaller than the size of the group of comm\_old, then some processes are returned MPI\_COMM\_NULL, in analogy to MPI\_CART\_CREATE and MPI\_COMM\_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI\_COMM\_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

**Example 8.2** Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

Then, the input arguments are:

nnodes = 
$$4$$
  
index =  $2, 3, 4, 6$   
edges =  $1, 3, 0, 3, 0, 2$ 

Thus, in C, index[0] is the degree of node zero, and index[i] - index[i-1] is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges[j], for  $0 \le j \le index[0] - 1$  and the list of neighbors of node i, i > 0, is stored in edges[j], index[i-1]  $\le j \le index[i] - 1$ . In Fortran, index(1) is the degree of node zero, and index(i+1) - index(i) is the degree

In Fortran, index(1) is the degree of hode zero, and index(1+1) - index(1) is the degree of node i, i=1, ..., nnodes-1; the list of neighbors of node zero is stored in edges(j), for  $1 \le j \le$  index(1) and the list of neighbors of node i, i > 0, is stored in edges(j), index(i)+1  $\le j \le$  index(i+1).

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be nonsymmetric.

Advice to users. Performance implications of using multiple edges or a nonsymmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (*End of advice to users.*)

Advice to implementors. The following topology information is likely to be stored with a communicator:

 $^{31}$ 

1 • Type of topology (Cartesian/graph), 2 • For a Cartesian topology: 3 1. ndims (number of dimensions), 4 2. dims (numbers of processes per coordinate direction), 53. periods (periodicity information), 6 7 4. own\_position (own position in grid, could also be computed from rank and 8 dims) 9 • For a graph topology: 10 1. index. 11 2. edges. 1213 which are the vectors defining the graph structure. 14For a graph structure the number of nodes is equal to the number of processes in 15the group. Therefore, the number of nodes does not have to be stored explicitly. 16An additional zero entry at the start of array index simplifies access to the topology 17information. (End of advice to implementors.) 18 19Distributed Graph Constructor 8.5.4 2021MPI\_GRAPH\_CREATE requires that each process passes the full (global) communication 22 graph to the call. This limits the scalability of this constructor. With the distributed graph 23interface, the communication graph is specified in a fully distributed fashion. Each process  $^{24}$ specifies only the part of the communication graph of which it is aware. Typically, this 25could be the set of processes from which the process will eventually receive or get data, 26or the set of processes to which the process will send or put data, or some combination of 27such edges. Two different interfaces can be used to create a distributed graph topology. 28

<sup>28</sup> MPI\_DIST\_GRAPH\_CREATE\_ADJACENT creates a distributed graph communicator with <sup>29</sup> each process specifying each of its incoming and outgoing (adjacent) edges in the logical <sup>30</sup> communication graph and thus requires minimal communication during creation.

MPI\_DIST\_GRAPH\_CREATE provides full flexibility such that any process can indicate that communication will occur between any pair of processes in the graph.

To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality of the reordering, and/or the time permitted for the MPI library to process the graph.

- 39
- 40
- 41 42
- 43
- 44
- 45
- 16
- 46 47
- 48

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT(comm\_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm\_dist\_graph)

	outdegree, destinations, a						
IN	comm_old	input communicator (handle)	$\frac{3}{4}$				
IN	indegree	size of sources and sourceweights arrays (non-negative	4 5				
		integer)	6				
IN	sources	ranks of processes for which the calling process is a	7				
		destination (array of non-negative integers)	8				
IN	sourceweights	weights of the edges into the calling process (array of	9				
IN	sourceweights	non-negative integers)	10				
		о о ,	11				
IN	outdegree	size of destinations and destweights arrays (non-negative integer)	12 13				
IN	destinations	ranks of processes for which the calling process is a	14				
		source (array of non-negative integers)	15				
IN	destweights	weights of the edges out of the calling process (array	16				
	5	of non-negative integers)	17 18				
IN	info	hints on optimization and interpretation of weights	18 19				
		(handle)	20				
IN	reorder	the ranks may be reordered (true) or not (false)	21				
		(logical)	22				
OUT	comm_dist_graph	communicator with distributed graph topology	23				
001	comm_dist_graph	(handle)	24				
		(nanalo)	25				
C binding	r		26				
		(MPI_Comm comm_old, int indegree,	27				
_	<b>0 1 0</b>	const int sourceweights[], int outdegree,	28 29				
	const int destination	ns[], const int destweights[],	30				
	MPI_Info info, int r	eorder, MPI_Comm *comm_dist_graph)	31				
Fortran 2	008 binding		32				
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights, <sup>33</sup>							

MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,	55
outdegree, destinations, destweights, info, reorder,	34
comm_dist_graph, ierror)	35
TYPE(MPI_Comm), INTENT(IN) :: comm_old	36
<pre>INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),</pre>	37
<pre>outdegree, destinations(outdegree), destweights(*)</pre>	38
TYPE(MPI_Info), INTENT(IN) :: info	39
LOGICAL, INTENT(IN) :: reorder	40
TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
	43
Fortran binding	44
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,	45

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT(COMM\_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, COMM\_DIST\_GRAPH, IERROR)  $\mathbf{2}$ 

## INTEGER COMM\_OLD, INDEGREE, SOURCES(\*), SOURCEWEIGHTS(\*), OUTDEGREE, DESTINATIONS(\*), DESTWEIGHTS(\*), INFO, COMM\_DIST\_GRAPH, IERROR LOGICAL REORDER

5MPI\_DIST\_GRAPH\_CREATE\_ADJACENT returns a handle to a new communicator 6 to which the distributed graph topology information is attached. Each process passes all 7 information about its incoming and outgoing edges in the virtual distributed graph topology. 8 The calling processes must ensure that each edge of the graph is described in the source 9 and in the destination process with the same weights. If there are multiple edges for a given 10 (source,dest) pair, then the sequence of the weights of these edges does not matter. The 11 complete communication topology is the combination of all edges shown in the sources arrays 12of all processes in **comm\_old**, which must be identical to the combination of all edges shown 13 in the destinations arrays. Source and destination ranks must be process ranks of comm\_old. 14This allows a fully distributed specification of the communication graph. Isolated processes 15(i.e., processes with no outgoing or incoming edges, that is, processes that have specified 16indegree and outdegree as zero and thus do not occur as source or destination rank in the 17graph specification) are allowed. 18

The call creates a new communicator comm\_dist\_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm\_dist\_graph is identical to the number of processes in comm\_old. The call to MPI\_DIST\_GRAPH\_CREATE\_ADJACENT is collective.

Weights are specified as non-negative integers and can be used to influence the process 23remapping strategy and other internal MPI optimizations. For instance, approximate count  $^{24}$ arguments of later communication calls along specific edges could be used as their edge 25weights. Multiplicity of edges can likewise indicate more intense communication between 26pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 27standard and is left to the implementation. In C or Fortran, an application can supply 28the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have 29 the same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some 30 but not all processes of comm\_old. If the graph is weighted but indegree or outdegree is  $^{31}$ zero, then MPI\_WEIGHTS\_EMPTY or any arbitrary array may be passed to sourceweights 32 or destweights respectively. Note that MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are 33 not special weight values; rather they are special values for the total array argument. In 34Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects like MPI\_BOTTOM (not 35 usable for initialization or assignment). See Section 2.5.4. 36

Advice to users. In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI\_UNWEIGHTED to the implementation. In this case MPI\_WEIGHTS\_EMPTY should be used instead. (*End of advice to users.*)

Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

Rationale. To ensure backward compatibility, MPI\_UNWEIGHTED may still be imple mented as NULL. See Annex B.3. (End of rationale.)

1

2

3

4

37

38

39

40

41

42 43

44

The meaning of the info and reorder arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n,	sources, deg	grees, destinations,	weights, info,
reorder, comm_dist_graph)			

IN	comm_old	input communicator (handle)	7
IN	n	number of source nodes for which this process	8
		specifies edges (non-negative integer)	9
IN	sources	array containing the <b>n</b> source nodes for which this	10
IIN	sources	process specifies edges (array of non-negative	11
		integers)	12
		- ,	13
IN	degrees	array specifying the number of destinations for each	14 15
		source node in the source node array (array of	16
		non-negative integers)	17
IN	destinations	destination nodes for the source nodes in the source	18
		node array (array of non-negative integers)	19
IN	weights	weights for source to destination edges (array of	20
		non-negative integers)	21
IN	info	hints on optimization and interpretation of weights	22
		(handle)	23
IN	reorder	the ranks may be reordered (true) or not (false)	24
	Teorder	(logical)	25
0.UT			26
OUT	comm_dist_graph	communicator with distributed graph topology	27
		added (handle)	28
~			29
C bindin	0		30
int MPI_	0 1	<pre>m comm_old, int n, const int sources[],</pre>	31
	0	<pre>const int destinations[],</pre>	32
	0	MPI_Info info, int reorder,	33 34
	MPI_Comm *comm_dist_	grahn)	34 35
Fortran	2008 binding		36
MPI_Dist	_graph_create(comm_old, n	, sources, degrees, destinations, weights,	37
	info roordor comm	dist graph iorror)	

info, reorder, comm\_dist\_graph, ierror) 38TYPE(MPI\_Comm), INTENT(IN) :: comm\_old 39INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(\*), 40weights(\*)  $^{41}$ TYPE(MPI\_Info), INTENT(IN) :: info 42LOGICAL, INTENT(IN) :: reorder 43TYPE(MPI\_Comm), INTENT(OUT) :: comm\_dist\_graph 44INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4546

# Fortran binding46MPI\_DIST\_GRAPH\_CREATE(COMM\_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,47INFO, REORDER, COMM\_DIST\_GRAPH, IERROR)48

1 2

INTEGER COMM\_OLD, N, SOURCES(\*), DEGREES(\*), DESTINATIONS(\*), WEIGHTS(\*), INFO, COMM\_DIST\_GRAPH, IERROR

LOGICAL REORDER

MPI\_DIST\_GRAPH\_CREATE returns a handle to a new communicator to which the 5distributed graph topology information is attached. Concretely, each process calls the con-6 structor with a set of directed (source.destination) communication edges as described below. 7 Every process passes an array of n source nodes in the sources array. For each source node, a 8 non-negative number of destination nodes is specified in the degrees array. The destination 9 nodes are stored in the corresponding consecutive segment of the destinations array. More 10 precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the 11 j-th such edge stored in destinations[degrees[0]+ $\dots$ +degrees[i-1]+j]. The weight of this edge 12is stored in weights[degrees[0]+ $\ldots$ +degrees[i-1]+i]. Both the sources and the destinations 13 arrays may contain the same node more than once, and the order in which nodes are listed 14as destinations or sources is not significant. Similarly, different processes may specify edges 15with the same source and destination nodes. Source and destination nodes must be pro-16cess ranks of comm\_old. Different processes may specify different numbers of source and 17destination nodes, as well as different source to destination edges. This allows a fully dis-18 tributed specification of the communication graph. Isolated processes (i.e., processes with 19no outgoing or incoming edges, that is, processes that do not occur as source or destination 20node in the graph specification) are allowed. 21

The call creates a new communicator comm\_dist\_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm\_dist\_graph is identical to the number of processes in comm\_old. The call to MPI\_DIST\_GRAPH\_CREATE is collective.

If reorder = false, all processes will have the same rank in comm\_dist\_graph as in comm\_old. If reorder = true then the MPI library is free to remap to other processes (of comm\_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

Weights are specified as non-negative integers and can be used to influence the process  $^{31}$ remapping strategy and other internal MPI optimizations. For instance, approximate count 32 arguments of later communication calls along specific edges could be used as their edge 33 weights. Multiplicity of edges can likewise indicate more intense communication between 34pairs of processes. However, the exact meaning of edge weights is not specified by the MPI 35 standard and is left to the implementation. In C or Fortran, an application can supply 36 the special value MPI\_UNWEIGHTED for the weight array to indicate that all edges have the 37 same (effectively no) weight. It is erroneous to supply MPI\_UNWEIGHTED for some but not 38 all processes of comm\_old. If the graph is weighted but n = 0, then MPI\_WEIGHTS\_EMPTY 39 or any arbitrary array may be passed to weights. Note that MPI\_UNWEIGHTED and 40 MPI\_WEIGHTS\_EMPTY are not special weight values; rather they are special values for the 41 total array argument. In Fortran, MPI\_UNWEIGHTED and MPI\_WEIGHTS\_EMPTY are objects 42like MPI\_BOTTOM (not usable for initialization or assignment). See Section 2.5.4. 43

Advice to users. In the case of an empty weights array argument passed while
 constructing a weighted graph, one should not pass NULL because the value of
 MPI\_UNWEIGHTED may be equal to NULL. The value of this argument would then
 be indistinguishable from MPI\_UNWEIGHTED to the implementation.

<sup>48</sup> MPI\_WEIGHTS\_EMPTY should be used instead. (*End of advice to users.*)

1

2

3

Advice to implementors. It is recommended that MPI\_UNWEIGHTED not be implemented as NULL. (End of advice to implementors.)

*Rationale.* To ensure backward compatibility, MPI\_UNWEIGHTED may still be implemented as NULL. See Annex B.3. (*End of rationale.*)

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

Advice to implementors. MPI implementations must document any additionally supported key-value info pairs. MPI\_INFO\_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI\_GRAPH\_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

**Example 8.3** As for Example 8.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

With MPI\_DIST\_GRAPH\_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

process	n	sources	degrees	destinations	weights
0	1	0	2	1,3	1,1
1	1	1	1	0	1
2	1	2	1	3	1
3	1	3	2	0,2	1,1

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

 $\overline{7}$ 

process	n	sources	degrees	destinations	weights
0	4	0,1,2,3	2,1,1,2	$1,\!3,\!0,\!3,\!0,\!2$	$1,\!1,\!1,\!1,\!1,\!1,\!1$
1	0	-	-	-	-
2	0	-	-	-	-
3	0	-	-	-	

In both cases above, the application could supply MPI\_UNWEIGHTED instead of explicitly providing identical weights.

MPI\_DIST\_GRAPH\_CREATE\_ADJACENT could be used to specify this graph using the following arguments:

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

**Example 8.4** A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled with Cartesian topologies, but can easily be captured with MPI\_DIST\_GRAPH\_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

```
/*
```

```
Input:
           dimensions P, Q
Condition: number of processes equal to P*Q; otherwise only
           ranks smaller than P*Q participate
*/
int rank, x, y;
int sources[1], degrees[1];
int destinations[8], weights[8];
MPI_Comm comm_dist_graph;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* get x and y dimension */
y=rank/P; x=rank%P;
/* get my communication partners along x dimension */
destinations [0] = P*y+(x+1)%P; weights [0] = 2;
destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;
/* get my communication partners along y dimension */
destinations[2] = P*((y+1))(Q)+x; weights[2] = 2;
destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;
/* get my communication partners along diagonals */
```

 $^{31}$ 

 $46 \\ 47$ 

```
destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;
sources[0] = rank;
degrees[0] = 8;
MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
weights, MPI_INFO_NULL, 1, &comm_dist_graph);
```

 $\mathbf{2}$ 

# 8.5.5 Topology Inquiry Functions

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

```
MPI_TOPO_TEST(comm, status)
```

IN	comm	communicator (handle)	18
OUT	status	topology type of communicator comm (state)	19
001	Status	topology type of communicator comm (state)	20
C bindin	α.		21
	s Copo_test(MPI_Comm comm, i	nt *status)	22 23
	-		23 24
	2008 binding		24 25
-	test(comm, status, ierror		26
	(MPI_Comm), INTENT(IN) ::		27
	ER, INTENT(OUT) :: status		28
INTE	GER, OPTIONAL, INTENT(OUT)	:: lerror	29
Fortran l	binding		30
MPI_TOPO	TEST(COMM, STATUS, IERROF		31
INTEC	GER COMM, STATUS, IERROR		32
The f	unction MPI TOPO TEST re	turns the type of topology that is assigned to a	33
communic		turns the type of topology that is assigned to a	34
	output value <b>status</b> is one of th	e following:	35
			36
MPI_GR/	APH	graph topology	37
MPI_CAI	RT	Cartesian topology	38
MPI_DIS	T_GRAPH	distributed graph topology	39
MPI_UN	DEFINED	no topology	40
			41
			42
			43
			44
			45

1 MPI\_GRAPHDIMS\_GET(comm, nnodes, nedges) 2 IN communicator for group with graph structure comm 3 (handle) 4 OUT nnodes number of nodes in graph (same as number of 5processes in the group) (integer) 6  $\overline{7}$ OUT nedges number of edges in graph (integer) 8 9 C binding 10 int MPI\_Graphdims\_get(MPI\_Comm comm, int \*nnodes, int \*nedges) 11 Fortran 2008 binding 12MPI\_Graphdims\_get(comm, nnodes, nedges, ierror) 13 TYPE(MPI\_Comm), INTENT(IN) :: comm 14INTEGER, INTENT(OUT) :: nnodes, nedges 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617Fortran binding 18 MPI\_GRAPHDIMS\_GET(COMM, NNODES, NEDGES, IERROR) 19INTEGER COMM, NNODES, NEDGES, IERROR 20The functions MPI\_GRAPHDIMS\_GET and MPI\_GRAPH\_GET retrieve the graph-topol-21ogy information that is associated with the communicator. The information provided by 22 MPI\_GRAPHDIMS\_GET can be used to dimension the vectors index and edges correctly for 23the following call to MPI\_GRAPH\_GET.  $^{24}$ 2526MPI\_GRAPH\_GET(comm, maxindex, maxedges, index, edges) 27IN communicator with graph structure (handle) 28comm 29IN maxindex length of vector index in the calling program (integer) 30 IN maxedges length of vector edges in the calling program (integer)  $^{31}$ OUT index array of integers containing the graph structure (for 32 details see the definition of MPI\_GRAPH\_CREATE) 33 34 OUT edges array of integers containing the graph structure 35 36 C binding 37 int MPI\_Graph\_get(MPI\_Comm comm, int maxindex, int maxedges, int index[], 38 int edges[]) 39 Fortran 2008 binding 4041MPI\_Graph\_get(comm, maxindex, maxedges, index, edges, ierror) 42TYPE(MPI\_Comm), INTENT(IN) :: comm INTEGER, INTENT(IN) :: maxindex, maxedges 43 INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges) 4445INTEGER, OPTIONAL, INTENT(OUT) :: ierror 46Fortran binding 47MPI\_GRAPH\_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) 48

1 INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(\*), EDGES(\*), IERROR  $\mathbf{2}$ 3 4 MPI\_CARTDIM\_GET(comm, ndims) 5 IN comm communicator with Cartesian structure (handle) 6 OUT ndims number of dimensions of the Cartesian structure 7 (integer) 9 10 C binding 11 int MPI\_Cartdim\_get(MPI\_Comm comm, int \*ndims) 12Fortran 2008 binding 13 MPI\_Cartdim\_get(comm, ndims, ierror) 14TYPE(MPI\_Comm), INTENT(IN) :: comm 15INTEGER, INTENT(OUT) :: ndims 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1718 Fortran binding 19 MPI\_CARTDIM\_GET(COMM, NDIMS, IERROR) 20INTEGER COMM, NDIMS, IERROR 21The functions MPI\_CARTDIM\_GET and MPI\_CART\_GET return the Cartesian topol-22 ogy information that is associated with the communicator. If comm is associated with a 23zero-dimensional Cartesian topology, MPI\_CARTDIM\_GET returns ndims = 0 and 24MPI\_CART\_GET will keep all output arguments unchanged. 252627MPI\_CART\_GET(comm, maxdims, dims, periods, coords) 28IN comm communicator with Cartesian structure (handle) 2930 IN length of vectors dims, periods, and coords in the maxdims 31calling program (integer) 32 OUT dims number of processes for each Cartesian dimension 33 (array of integers) 34 OUT periods periodicity (true/false) for each Cartesian dimension 35 (array of logicals) 36 37 OUT coords coordinates of calling process in Cartesian structure 38 (array of integers) 39 40 C binding 41 int MPI\_Cart\_get(MPI\_Comm comm, int maxdims, int dims[], int periods[], 42int coords[]) 43 Fortran 2008 binding 44MPI\_Cart\_get(comm, maxdims, dims, periods, coords, ierror) 45TYPE(MPI\_Comm), INTENT(IN) :: comm 46INTEGER, INTENT(IN) :: maxdims 47INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims) 48

```
1
          LOGICAL, INTENT(OUT) :: periods(maxdims)
\mathbf{2}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
      Fortran binding
4
     MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
5
          INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
6
          LOGICAL PERIODS(*)
7
8
9
     MPI_CART_RANK(comm, coords, rank)
10
11
       IN
                                              communicator with Cartesian structure (handle)
                 comm
12
       IN
                 coords
                                              integer array (of size ndims) specifying the Cartesian
13
                                              coordinates of a process
14
       OUT
                 rank
                                              rank of specified process (integer)
15
16
17
     C binding
18
      int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
19
     Fortran 2008 binding
20
     MPI_Cart_rank(comm, coords, rank, ierror)
21
          TYPE(MPI_Comm), INTENT(IN) :: comm
22
          INTEGER, INTENT(IN) :: coords(*)
23
          INTEGER, INTENT(OUT) :: rank
^{24}
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     Fortran binding
27
     MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
28
          INTEGER COMM, COORDS(*), RANK, IERROR
29
          For a communicator with an associated Cartesian topology, the function
30
      MPI_CART_RANK translates the logical process coordinates to process ranks. For dimen-
^{31}
     sion i with periods(i) = true, if the coordinate, coords(i), is out of range, that is, coords(i) < 0
32
      or coords(i) > dims(i), it is shifted back to the interval 0 < coords(i) < dims(i) automatically.
33
      Out-of-range coordinates are erroneous for nonperiodic dimensions.
34
          If comm is associated with a zero-dimensional Cartesian topology, coords is not signif-
35
      icant and 0 is returned in rank.
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_CART	_COORDS(comm, rank, maxdi	ms, coords)	1		
IN	comm	communicator with Cartesian structure (handle)	2 3		
IN	rank	rank of a process within group of $comm\ (\mathrm{integer})$	4		
IN	maxdims	length of vector <b>coords</b> in the calling program (integer)	5 6		
OUT	coords	integer array (of size maxdims) containing the Cartesian coordinates of specified process (array of integers)	7 8 9 10		
C binding		<pre>int rank, int maxdims, int coords[])</pre>	11 12		
			13 14		
MPI_Cart_c TYPE(N INTEGH	Fortran 2008 binding       14         MPI_Cart_coords(comm, rank, maxdims, coords, ierror)       15         TYPE(MPI_Comm), INTENT(IN) :: comm       16         INTEGER, INTENT(IN) :: rank, maxdims       17         INTEGER, INTENT(OUT) :: coords(maxdims)       18         INTEGER, OPTIONAL, INTENT(OUT) :: ierror       19				
MPI_CART_C	Fortran binding       20         MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)       22         INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR       23				
MPI_CART		linates translation is provided by ated with a zero-dimensional Cartesian topology,	24 25 26 27 28		
MPI_GRAP	H_NEIGHBORS_COUNT(com	m, rank, nneighbors)	29		
IN	comm	communicator with graph topology (handle)	30		
IN	rank	rank of process in group of comm (integer)	31 32		
OUT	nneighbors	number of neighbors of specified process (integer)	33 34		
C binding int MPI_G		Comm comm, int rank, int *nneighbors)	35 36		
Fortran 2008 binding MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror) TVPE(MPI_Comm) INTENT(IN) :: comm					
INTEGE	INTEGER, INTENT(IN) :: rank 41 INTEGER, INTENT(OUT) :: nneighbors 42				
INTEGER, OPTIONAL, INTENT(UOT) :: lerror       43         Fortran binding       44         MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)       45         INTEGER COMM, RANK, NNEIGHBORS, IERROR       46         47       47					

```
1
     MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)
2
       IN
                 comm
                                              communicator with graph topology (handle)
3
       IN
                 rank
                                             rank of process in group of comm (integer)
4
5
       IN
                 maxneighbors
                                             size of array neighbors (integer)
6
       OUT
                 neighbors
                                             ranks of processes that are neighbors to specified
7
                                             process (array of integers)
8
9
     C binding
10
     int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
11
                     int neighbors[])
12
13
     Fortran 2008 binding
14
     MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
15
          TYPE(MPI_Comm), INTENT(IN) :: comm
16
          INTEGER, INTENT(IN) :: rank, maxneighbors
17
          INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
18
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     Fortran binding
20
     MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
21
          INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
22
23
          MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency
^{24}
     information for a graph topology. The returned count and array of neighbors for the queried
25
     rank will both include all neighbors and reflect the same edge ordering as was specified by
26
     the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT
27
     and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array
28
     passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1]
29
     is zero):
30
31
         • The number of neighbors (nneighbors) returned from
32
           MPI_GRAPH_NEIGHBORS_COUNT will be (index[rank] - index[rank-1]).
33
         • The neighbors array returned from MPI_GRAPH_NEIGHBORS will be
34
           edges[index[rank-1]] through edges[index[rank]-1].
35
36
37
      Example 8.5 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix
38
      (note that some neighbors are listed multiple times):
39
40
                                                  neighbors
                                        process
41
                                                  1, 1, 3
                                           0
42
                                                  0,0
                                           1
43
                                           \mathbf{2}
                                                  3
44
                                           3
                                                  0, 2, 2
45
46
      Thus, the input arguments to MPI_GRAPH_CREATE are:
47
48
```

nnodes =	4
index =	3,5,6,9
edges =	1, 1, 3, 0, 0, 3, 0, 2, 2

Therefore, calling MPI\_GRAPH\_NEIGHBORS\_COUNT and MPI\_GRAPH\_NEIGHBORS for each of the 4 processes will return:

Input rank	Count	Neighbors
0	3	1, 1, 3
1	2	0, 0
2	1	3
3	3	0, 2, 2

**Example 8.6** Suppose that comm is a communicator with a shuffle-exchange topology. The group has  $2^n$  members. Each process is labeled by  $a_1, \ldots, a_n$  with  $a_i \in \{0, 1\}$ , and has three neighbors: exchange $(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$  ( $\bar{a} = 1 - a$ ), shuffle $(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$ , and unshuffle $(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$ . The graph adjacency list is illustrated below for n = 3.

1	node	exchange	shuffle	unshuffle
		neighbors(1)	neighbors(2)	neighbors(3)
0	(000)	1	0	0
1	(001)	0	2	4
2	(010)	3	4	1
3	(011)	2	6	5
4	(100)	5	1	2
5	(101)	4	3	6
6	(110)	7	5	3
7	(111)	6	7	7

Suppose that the communicator **comm** has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

	410		CHAPTER 8. PROCESS TOPOLOGIES	
1 2 3			HBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS pro- r a distributed graph topology.	
4 5	MPI_DIS	T_GRAPH_NEIGHBC	RS_COUNT(comm, indegree, outdegree, weighted)	
6 7	IN	comm	communicator with distributed graph topology (handle)	
8 9 10	OUT	indegree	number of edges into this process (non-negative integer)	
10 11 12	OUT	outdegree	number of edges out of this process (non-negative integer)	
13 14 15	OUT	weighted	false if MPI_UNWEIGHTED was supplied during creation, true otherwise (logical)	
16 17 18 19	C bindin int MPI_	_Dist_graph_neighb	ors_count(MPI_Comm comm, int *indegree, ree, int *weighted)	
20 21 22 23 24 25	MPI_Dist TYPI INTI LOGJ	E(MPI_Comm), INTEN EGER, INTENT(OUT) ICAL, INTENT(OUT)	:: indegree, outdegree	
26 27 28 29 30	MPI_DIST INTH		COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR) E, OUTDEGREE, IERROR	
31 32 33				
34 35				
36				
37 38				
39				
40 41				
41 42				
43				
44				
45				
46				
47				
48				

MPI_DIST		omm, maxindegree, sources, sourceweights, nations, destweights)	1 2	
IN	comm	communicator with distributed graph topology (handle)	3 4 5	
IN	maxindegree	size of sources and sourceweights arrays (non-negative integer)	5 6 7	
OUT	sources	processes for which the calling process is a destination (array of non-negative integers)	8 9	
OUT	sourceweights	weights of the edges into the calling process (array of non-negative integers)	10 11	
IN	maxoutdegree	size of destinations and destweights arrays (non-negative integer)	12 13 14	
OUT	destinations	processes for which the calling process is a source (array of non-negative integers)	15 16	
OUT	destweights	weights of the edges out of the calling process (array of non-negative integers)	17 18 19	
Fortran	int sourceweight int destweights[ 2008 binding		22 23 24 25	
TYPE INTE INTE INTE	maxoutdegree, de (MPI_Comm), INTENT(IN)	<pre>xindegree, maxoutdegree ources(maxindegree), xoutdegree) *), destweights(*)</pre>	26 27 28 29 30 31 32 33	
Fortran binding MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR				
These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (poten- tially by processes other than the calling process in the case of ADI DIST_CRAPH_CREATE) Multiple 1 Counter the second				

MPI\_DIST\_GRAPH\_CREATE). Multiply-defined edges are all counted and returned by
 MPI\_DIST\_GRAPH\_NEIGHBORS in some order. If MPI\_UNWEIGHTED is supplied for
 sourceweights or destweights or both, or if MPI\_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays.
 If the communicator was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT then for

each rank in comm, the order of the values in sources and destinations is identical to the in put that was used by the process with the same rank in comm\_old in the creation call. If the
 communicator was created with MPI\_DIST\_GRAPH\_CREATE then the only requirement on
 the order of values in sources and destinations is that two calls to the routine with same in put argument comm will return the same sequence of edges. If maxindegree or maxoutdegree
 is smaller than the numbers returned by MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT, then
 only the first part of the full list is returned.

8 9

10

11

12

Advice to implementors. Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI\_DIST\_GRAPH\_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (End of advice to implementors.)

13 14 15

8.5.6 Cartesian Shift Coordinates

<sup>16</sup> <sup>17</sup> If the process topology is a Cartesian structure, an MPI\_SENDRECV operation may be used <sup>18</sup> along a coordinate direction to perform a shift of data. As input, MPI\_SENDRECV takes <sup>19</sup> the rank of a source process for the receive, and the rank of a destination process for the <sup>20</sup> send. If the function MPI\_CART\_SHIFT is called for a Cartesian process group, it provides <sup>21</sup> the calling process with the above identifiers, which then can be passed to MPI\_SENDRECV. <sup>22</sup> The user specifies the coordinate direction and the size of the step (positive or negative, <sup>23</sup> but not zero). The function is local.

24 25

MPI\_CART\_SHIFT(comm, direction, disp, rank\_source, rank\_dest)

```
26
       IN
                 comm
                                             communicator with Cartesian structure (handle)
27
       IN
                 direction
                                             coordinate dimension of shift (integer)
28
29
       IN
                 disp
                                             displacement (> 0: upwards shift, < 0: downwards
30
                                             shift) (integer)
^{31}
       OUT
                 rank_source
                                             rank of source process (integer)
32
       OUT
33
                 rank_dest
                                             rank of destination process (integer)
34
35
     C binding
36
     int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
37
                    int *rank_source, int *rank_dest)
38
     Fortran 2008 binding
39
     MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
40
          TYPE(MPI_Comm), INTENT(IN) :: comm
41
          INTEGER, INTENT(IN) :: direction, disp
42
          INTEGER, INTENT(OUT) :: rank_source, rank_dest
43
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     Fortran binding
46
     MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
47
          INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
48
```

The direction argument indicates the coordinate dimension to be traversed by the shift. The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.

Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI\_CART\_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI\_PROC\_NULL may be returned in rank\_source or rank\_dest, indicating that the source or the destination for the shift is out of range.

It is erroneous to call MPI\_CART\_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI\_CART\_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

**Example 8.7** The communicator, **comm**, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of **REALs** is stored one element per process, in variable **A**. One wishes to skew this array, by shifting column **i** (vertically, i.e., along the column) by **i** steps.

Advice to users. In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (End of advice to users.)

### 8.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(comm, remain_dims, newcomm)				
	,	,	36	
IN	comm	communicator with Cartesian structure (handle)	37	
IN	remain_dims	the i-th entry of remain_dims specifies whether the	38	
		i-th dimension is kept in the subgrid (true) or is	39	
		dropped (false) (array of logicals)	40	
OUT	newcomm	communicator containing the subgrid that includes	41	
		the calling process (handle)	42	
			43	
Chindin	~		44	
C binding				

<pre>int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcome </pre>	omm) $46$
Fortran 2008 binding	47
MPI_Cart_sub(comm, remain_dims, newcomm, ierror)	48

 $^{24}$ 

_	
1	TYPE(MPI_Comm), INTENT(IN) :: comm
2	LOGICAL, INTENT(IN) :: remain_dims(*)
3	TYPE(MPI_Comm), INTENT(OUT) :: newcomm
4	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5	INTEGER, OFITONAL, INTENT(OOI) TETTOT
6	Fortran binding
	MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
7	INTEGER COMM, NEWCOMM, IERROR
8	LOGICAL REMAIN_DIMS(*)
9	LOGICAL REMAIN_DIND(*)
10	MPI_CART_SUB can be used to partition the group associated with a communica-
11	tor that has an associated Cartesian topology into subgroups that form lower-dimensional
12	Cartesian subgrids, and to build for each subgroup a communicator with the associated sub-
13	grid Cartesian topology. The topologies of the new communicators describe the subgrids.
14	The number of dimensions of the subgrids is the number of remaining dimensions, i.e., the
15	
16	number of true values in remain_dims. The numbers of MPI processes in each coordinate
17	direction of the subgrids are the remaining numbers of MPI processes in each coordinate di-
	rection of the grid associated with the original communicator, i.e., the values of the original
18	grid dimensions for which the corresponding entry in remain_dims is true. The periodic-
19	ity for the remaining dimensions in the new communicator is preserved from the original
20	communicator. If all entries in remain_dims are false or comm is already associated with a
21	zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional
22	Cartesian topology. (This function is closely related to MPI_COMM_SPLIT.)
23	
24	<b>Example 8.8</b> Assume that MPI_Cart_create(, comm) has defined a $(2 \times 3 \times 4)$ grid. Let
25	remain_dims = (true, false, true). Then a call to
26	
27	MPI_Cart_sub(comm, remain_dims, newcomm)
28	
29	will create three communicators each with eight processes in a $2 \times 4$ Cartesian topology. If
30	$remain\_dims = (false, false, true) $ then the call to
31	
	MDI Contauto (comme normaline dimensional)
32	MPI_Cart_sub(comm, remain_dims, newcomm)
33	will create six non-overlapping communicators, each with four processes, in a one-
33 34	
	will create six non-overlapping communicators, each with four processes, in a one-
34	will create six non-overlapping communicators, each with four processes, in a one- dimensional Cartesian topology.
34 35	will create six non-overlapping communicators, each with four processes, in a one-
34 35 36	<ul><li>will create six non-overlapping communicators, each with four processes, in a one- dimensional Cartesian topology.</li><li>8.5.8 Low-Level Topology Functions</li></ul>
34 35 36 37	<ul> <li>will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.</li> <li>8.5.8 Low-Level Topology Functions</li> <li>The two additional functions introduced in this section can be used to implement all other</li> </ul>
34 35 36 37 38	<ul> <li>will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.</li> <li>8.5.8 Low-Level Topology Functions</li> <li>The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, except when</li> </ul>
34 35 36 37 38 39 40	<ul> <li>will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.</li> <li>8.5.8 Low-Level Topology Functions</li> <li>The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, except when creating additional virtual topology capabilities other than those provided by MPI. The two</li> </ul>
34 35 36 37 38 39 40 41	<ul> <li>will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.</li> <li>8.5.8 Low-Level Topology Functions</li> <li>The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, except when</li> </ul>
34 35 36 37 38 39 40	<ul> <li>will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.</li> <li>8.5.8 Low-Level Topology Functions</li> <li>The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, except when creating additional virtual topology capabilities other than those provided by MPI. The two</li> </ul>

MPI_0	CART_MAP(comm, ndims	s, dims, periods, newrank)	1	
IN	comm	input communicator (handle)	2	
IN	ndims	number of dimensions of Cartesian structure (integer)	3 4	
IN	dims	integer array of size <b>ndims</b> specifying the number of processes in each coordinate direction	5 6	
IN	periods	logical array of size <b>ndims</b> specifying the periodicity specification in each coordinate direction	7 8 9	
OUT	- newrank	reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (integer)	10 11 12	
C bir	ding		13 14	
		comm, int ndims, const int dims[],	15	
	const int per	riods[], int *newrank)	16	
Fortr	an 2008 binding		17	
	•	dims, periods, newrank, ierror)	18 19	
Т	YPE(MPI_Comm), INTENT	C(IN) :: comm	20	
	NTEGER, INTENT(IN) ::		21	
	OGICAL, INTENT(IN) ::	-	22	
	INTEGER, INTENT(OUT) :: newrank			
T	INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
Fortran binding 25				
	MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) <sup>2</sup>			
	INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR			
L	LOGICAL PERIODS(*)			
Ν	IPI_CART_MAP compute	es an "optimal" placement for the calling process on the phys-	29 30	
ical m	achine. A possible impler	mentation of this function is to always return the rank of the	31	
calling	g process, that is, not to p	perform any reordering.	32	
			33	
	Advice to implementors.		34	
		art), with reorder = true can be implemented by calling $dime_{n}$ dime_nericals neuronly, then calling	35	
		ndims, dims, periods, newrank), then calling m, color, key, comm_cart), with color = 0 if newrank $\neq$	36	
		= MPI_UNDEFINED otherwise, and key = newrank. If ndims	37	
	,	sional Cartesian topology is created.	38	
			39	
		_SUB(comm, remain_dims, comm_new) can be implemented _SPLIT(comm, color, key, comm_new), using a single number	40	
	-	nsions as color and a single number encoding of the preserved	41	
	dimensions as key.	insides as color and a bingle number encoding of the preserved	42 43	
	-	ony functions can be implemented locally using the targle me	43 44	
		ogy functions can be implemented locally, using the topology d with the communicator. ( <i>End of advice to implementors.</i> )	45	
	mormation that is calle	a with the communicator. (Drid of addice to implementations.)	46	

The corresponding function for graph structures is as follows.

```
1
     MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)
2
       IN
                 comm
                                             input communicator (handle)
3
       IN
                 nnodes
                                             number of graph nodes (integer)
4
5
       IN
                 index
                                             integer array specifying the graph structure, see
6
                                             MPI_GRAPH_CREATE
7
       IN
                 edges
                                             integer array specifying the graph structure
8
       OUT
                 newrank
                                             reordered rank of the calling process;
9
                                             MPI_UNDEFINED if the calling process does not
10
                                             belong to graph (integer)
11
12
     C binding
13
14
     int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],
                    const int edges[], int *newrank)
15
16
     Fortran 2008 binding
17
     MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
18
          TYPE(MPI_Comm), INTENT(IN) :: comm
19
          INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
20
          INTEGER, INTENT(OUT) :: newrank
21
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     Fortran binding
^{24}
     MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
25
          INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
26
27
           Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index,
28
           edges, reorder, comm_graph), with reorder = true can be implemented by calling
29
           MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling
30
           MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank \neq
31
           MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.
32
           All other graph topology functions can be implemented locally, using the topology
33
           information that is cached with the communicator. (End of advice to implementors.)
34
35
36
            Neighborhood Collective Communication on Process Topologies
     8.6
37
38
     MPI process topologies specify a communication graph, but they implement no commu-
39
     nication function themselves. Many applications require sparse nearest neighbor commu-
40
     nications that can be expressed as graph topologies. We now describe several collective
```

operations that perform communication along the edges of a process topology. All of these

functions are collective; i.e., they must be called by all processes in the specified commu-

nicator. See Section 6 for an overview of other dense (global) collective communication

and destinations containing  $0, \ldots, n-1$ , where n is the number of processes in the group

of comm\_old (i.e., the graph is fully connected and also includes an edge from each node

to itself), then the sparse neighborhood communication routine performs the same data

If the graph was created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT with sources

operations and the semantics of collective operations.

41

42

43

44

45

46

47

exchange as the corresponding dense (fully-connected) collective operation. In the case of a Cartesian communicator, only nearest neighbor communication is provided, corresponding to rank\_source and rank\_dest in MPI\_CART\_SHIFT with input disp = 1.

*Rationale.* Neighborhood collective communications enable communication on a process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [39]. This functionality can significantly simplify the implementation of neighbor exchanges [35]. (*End of rationale.*)

For a distributed graph topology, created with MPI\_DIST\_GRAPH\_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined as the sequence returned by MPI\_DIST\_GRAPH\_NEIGHBORS for destinations and sources, respectively. For a general graph topology, created with MPI\_GRAPH\_CREATE, the use of neighborhood collective communication is restricted to adjacency matrices, where the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix). In this case, the order of neighbors in the send and receive buffers is defined as the sequence of neighbors as returned by MPI\_GRAPH\_NEIGHBORS. Note that general graph topologies should generally be replaced by the distributed graph topologies.

For a Cartesian topology, created with MPI\_CART\_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined by order of the dimensions, first the neighbor in the negative direction and then in the positive direction with displacement 1. The numbers of sources and destinations in the communication routines are 2\*ndims with ndims defined in MPI\_CART\_CREATE. If a neighbor does not exist, i.e., at the border of a Cartesian topology in the case of a nonperiodic virtual grid dimension (i.e., periods[...]==false), then this neighbor is defined to be MPI\_PROC\_NULL.

If a neighbor in any of the functions is MPI\_PROC\_NULL, then the neighborhood collective communication behaves like a point-to-point communication with MPI\_PROC\_NULL in this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

### 8.6.1 Neighborhood Gather

In this function, each process i gathers data items from each process j if an edge (j, i) exists in the topology graph, and each process i sends the same data items to all processes j where an edge (i, j) exists. The send buffer is sent to each neighboring process and the l-th block in the receive buffer is received from the l-th neighbor.  $\mathbf{5}$ 

 $\overline{7}$ 

 $^{24}$ 

```
1
     MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
\mathbf{2}
                    comm)
3
       IN
                sendbuf
                                            starting address of send buffer (choice)
4
       IN
                sendcount
                                            number of elements sent to each neighbor
5
                                            (non-negative integer)
6
7
       IN
                sendtype
                                            datatype of send buffer elements (handle)
8
       OUT
                recvbuf
                                            starting address of receive buffer (choice)
9
       IN
                 recvcount
                                            number of elements received from each neighbor
10
                                            (non-negative integer)
11
12
       IN
                 recvtype
                                            datatype of receive buffer elements (handle)
13
       IN
                                            communicator with topology structure (handle)
                comm
14
15
     C binding
16
     int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
17
                    MPI_Datatype sendtype, void *recvbuf, int recvcount,
18
                    MPI_Datatype recvtype, MPI_Comm comm)
19
20
     int MPI_Neighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,
21
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
22
                    MPI_Datatype recvtype, MPI_Comm comm)
23
     Fortran 2008 binding
24
     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
25
                    recvtype, comm, ierror)
26
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
27
         INTEGER, INTENT(IN) :: sendcount, recvcount
28
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
         TYPE(*), DIMENSION(..) :: recvbuf
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
34
                    recvtype, comm, ierror) !(_c)
35
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
37
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
38
         TYPE(*), DIMENSION(...) :: recvbuf
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     Fortran binding
42
     MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
43
                    RECVTYPE, COMM, IERROR)
44
          <type> SENDBUF(*), RECVBUF(*)
45
          INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
46
47
48
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 8.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
outdegree, dsts, MPI_UNWEIGHTED);
int k:
```

int k;

```
/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
   MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],...);</pre>
```

```
MPI_Waitall(...);
```

Figure 8.1 shows the neighborhood gather communication of one process with outgoing neighbors  $d_0 \ldots d_3$  and incoming neighbors  $s_0 \ldots s_5$ . The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.

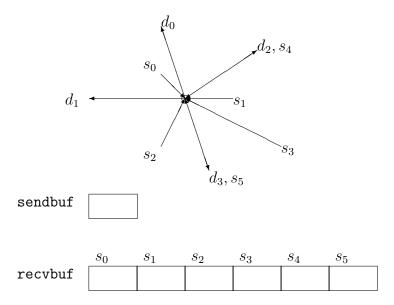


Figure 8.1: Neighborhood gather communication example

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies 48

 $\mathbf{2}$ 

 $^{24}$ 

 $45 \\ 46$ 

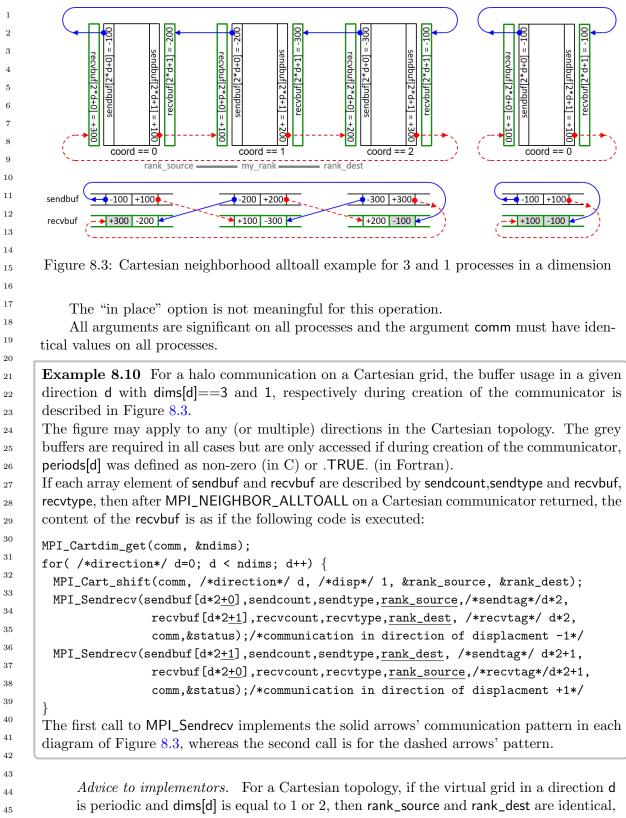
1 2	sendbuf	100	200	€ 300 €		
3	recvbuf	300 200	100 300	200 10	0	-+ 100 100
4 5 6	Figure 8.2	: Cartesian neig	hborhood allga	ther example for 3	and 1 proces	ses in a dimension
7 8 9 10	that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.					
11 12 13		-		ons, the same type ph is connected or	-	required indepen- <i>rationale.</i> )
14 15	The '	'in place" option	ı is not meanin	gful for this operation	tion.	
16 17 18 19 20 21	dims[d]== ure 8.2. The figure buffers are	=3 and 1, respective e may apply to the required in all of the section of the sec	ctively during any (or multip cases but are or	creation of the con- le) directions in th	mmunicator is ne Cartesian t ng creation of	en direction <b>d</b> with s described in Fig- copology. The grey the communicator,
23 24 25 26 27	The vector variant of MPI_NEIGHBOR_ALLGATHER allows one to gather different numbers of elements from each neighbor. MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)					
28 29	IN	sendbuf		starting address of	send buffer (ch	noice)
30 31	IN	sendcount		number of elements (non-negative integ		neighbor
32 33	IN	sendtype		datatype of send b	uffer elements	(handle)
34	OUT	recvbuf		starting address of	receive buffer	(choice)
35 36 37	IN	recvcounts		non-negative intege containing the num from each neighbor	nber of element	,
38 39 40	IN	displs		integer array (of let the displacement (n place the incoming	relative to recv	buf) at which to
41 42	IN	recvtype		datatype of receive	buffer element	s (handle)
43	IN	comm		communicator with	n topology stru	cture (handle)
44 45	C hindin	œ				
46	C bindin int MPI_1	-	therv(const	void *sendbuf,	int sendcou	nt,
47				e, void *recvbuf		
48		const int	t displs[], N	<pre>IPI_Datatype rec</pre>	vtype, MPI_	Comm comm)

```
1
int MPI_Neighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,
                                                                                     2
              MPI_Datatype sendtype, void *recvbuf,
                                                                                     3
              const MPI_Count recvcounts[], const MPI_Aint displs[],
              MPI_Datatype recvtype, MPI_Comm comm)
                                                                                     4
                                                                                     5
Fortran 2008 binding
                                                                                     6
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                     7
              displs, recvtype, comm, ierror)
                                                                                     8
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                     9
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
                                                                                     10
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                     11
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                     12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     14
                                                                                     15
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                     16
              displs, recvtype, comm, ierror) !(_c)
                                                                                     17
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                    18
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
                                                                                     19
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                    20
                                                                                    21
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                    22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     24
Fortran binding
                                                                                     25
MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
                                                                                     26
              DISPLS, RECVTYPE, COMM, IERROR)
                                                                                    27
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    28
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
                                                                                    29
               IERROR
                                                                                    30
                                                                                    31
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                    32
graph communicators as described in Section 8.6. If comm is a distributed graph commu-
                                                                                    33
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
                                                                                    34
and receives from each of its incoming neighbors:
                                                                                    35
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
                                                                                    36
                                                                                    37
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                     38
                                                                                     39
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                           outdegree, dsts, MPI_UNWEIGHTED);
                                                                                     40
                                                                                     41
int k;
                                                                                     42
/* assume sendbuf and recvbuf are of type (char*) */
                                                                                     43
                                                                                     44
for(k=0; k<outdegree; ++k)</pre>
  MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);
                                                                                     45
                                                                                     46
                                                                                     47
for(k=0; k<indegree; ++k)</pre>
  MPI_Irecv(recvbuf+displs[k]*extent(recvtype), recvcounts[k], recvtype,
                                                                                     48
```

1		<pre>srcs[k],);</pre>	
3	MPI_Waita	all();	
4 5 7 8 9 10 11 12 13	to the typ srcs[l] == j data data receive between see placed into The " All ar	be signature associated with This implies that the amoved, pairwise between every p ender and receiver are still all precvbuf beginning at offset in place" option is not mean	th sendcount, sendtype, at process $j$ must be equal a recvcounts[I], recvtype at any other process with bunt of data sent must be equal to the amount of air of communicating processes. Distinct type maps llowed. The data received from the l-th neighbor is displs[I] elements (in terms of the recvtype). ingful for this operation. I processes and the argument comm must have iden-
14 15	8.6.2 Ne	ighbor Alltoall	
16 17 18 19 20 21 22	In this function, each process $i$ receives data items from each process $j$ if an edge $(j, i)$ exists in the topology graph or Cartesian topology. Similarly, each process $i$ sends data items to all processes $j$ where an edge $(i, j)$ exists. This call is more general than MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. The $k$ -th block in send buffer is sent to the $k$ -th neighboring process and the $l$ -th block in the receive buffer is received from the $l$ -th neighbor.		
23 24 25	MPI_NEIG	HBOR_ALLTOALL(sendbuf, s comm)	sendcount, sendtype, recvbuf, recvcount, recvtype,
26	IN	sendbuf	starting address of send buffer (choice)
27 28 29	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
30	IN	sendtype	datatype of send buffer elements (handle)
31	OUT	recvbuf	starting address of receive buffer (choice)
32 33 34	IN	recvcount	number of elements received from each neighbor (non-negative integer)
35	IN	recvtype	datatype of receive buffer elements (handle)
36	IN	comm	communicator with topology structure (handle)
37 38 39	C binding int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,		
40 41 42		MPI_Datatype sendty MPI_Datatype recvty	pe, void *recvbuf, int recvcount, pe, MPI_Comm comm)
43 44 45	int MPI_N	•	t void *sendbuf, MPI_Count sendcount, pe, void *recvbuf, MPI_Count recvcount, pe, MPI_Comm comm)
46 47 48		2008 binding bor_alltoall(sendbuf, se recvtype, comm, ier:	endcount, sendtype, recvbuf, recvcount, ror)

```
1
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                      2
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                      3
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                      4
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      5
                                                                                      6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      7
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                      8
              recvtype, comm, ierror) !(_c)
                                                                                      9
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                      10
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                      11
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                      12
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                      13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      15
                                                                                      16
Fortran binding
                                                                                      17
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                      18
              RECVTYPE, COMM, IERROR)
                                                                                      19
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
                                                                                      20
                                                                                      21
    This function supports Cartesian communicators, graph communicators, and distributed
                                                                                      22
graph communicators as described in Section 8.6. If comm is a distributed graph commu-
                                                                                      23
nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
                                                                                      24
and receives from each of its incoming neighbors:
                                                                                      25
                                                                                      26
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
                                                                                      27
int *srcs=(int*)malloc(indegree*sizeof(int));
                                                                                      28
int *dsts=(int*)malloc(outdegree*sizeof(int));
                                                                                      29
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                                                                                      30
                           outdegree, dsts, MPI_UNWEIGHTED);
                                                                                      31
int k;
                                                                                      32
                                                                                      33
/* assume sendbuf and recvbuf are of type (char*) */
                                                                                      34
for(k=0; k<outdegree; ++k)</pre>
                                                                                      35
  MPI_Isend(sendbuf+k*sendcount*extent(sendtype), sendcount, sendtype,
                                                                                      36
             dsts[k],...);
                                                                                      37
                                                                                      38
for(k=0; k<indegree; ++k)</pre>
                                                                                      39
  MPI_Irecv(recvbuf+k*recvcount*extent(recvtype), recvcount, recvtype,
                                                                                      40
             srcs[k],...);
                                                                                      41
                                                                                      42
MPI_Waitall(...);
                                                                                      43
                                                                                      44
```

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.



is periodic and dims[d] is equal to 1 or 2, then rank\_source and rank\_dest are identical, but still all ndims send and ndims receive operations use different buffers. If in this case, the two send and receive operations per direction or of all directions are internally parallelized, then the several send and receive operations for the same sender-receiver

46

47

process pair shall be initiated in the same sequence on sender and receiver side or they shall be distinguished by different tags. The code above shows a valid sequence of operations and tags. (*End of advice to implementors.*)

The vector variant of MPI\_NEIGHBOR\_ALLTOALL allows sending/receiving different numbers of elements to and from each neighbor.

MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,					
	rdispls, recvtype, comm)		10		
IN	sendbuf	starting address of send buffer (choice)	11		
IN	sendcounts	non-negative integer array (of length outdegree)	12		
	Sendeounts	specifying the number of elements to send to each	13		
		neighbor	14		
		neignoor	15		
IN	sdispls	integer array (of length outdegree). Entry j specifies	16		
		the displacement (relative to sendbuf) from which to	17		
		send the outgoing data to neighbor j	18		
IN	sendtype	datatype of send buffer elements (handle)	19		
OUT	recvbuf	starting address of receive buffer (choice)	20		
			21		
IN	recvcounts	non-negative integer array (of length indegree)	22		
		specifying the number of elements that are received	23		
		from each neighbor	24		
IN	rdispls	integer array (of length indegree). Entry i specifies	25		
		the displacement (relative to $recvbuf$ ) at which to	26		
		place the incoming data from neighbor i	27		
IN	recvtype	datatype of receive buffer elements (handle)	28		
	i cevtype	· -	29		
IN	comm	communicator with topology structure (handle)	30		
			31		

# C binding

<pre>int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[]</pre>		
<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>	34	
<pre>const int recvcounts[], const int rdispls[],</pre>	35	
MPI_Datatype recvtype, MPI_Comm comm)	36	
int MPI_Neighbor_alltoallv_c(const void *sendbuf,	37	
•		
<pre>const MPI_Count sendcounts[], const MPI_Aint sdispls[],</pre>	39	
MPI_Datatype sendtype, void *recvbuf,	40	
<pre>const MPI_Count recvcounts[], const MPI_Aint rdispls[],</pre>	41	
MPI_Datatype recvtype, MPI_Comm comm)	42	
Fortran 2008 binding		
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,		
recvcounts, rdispls, recvtype, comm, ierror)	45	
TYPE(*), DIMENSION(), INTENT(IN) :: sendbuf	46	
<pre>INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),</pre>	47	
rdispls(*)	48	

 $\mathbf{2}$ 

 $\mathbf{5}$ 

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
\mathbf{2}
         TYPE(*), DIMENSION(..) :: recvbuf
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
6
                   recvcounts, rdispls, recvtype, comm, ierror) !(_c)
7
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
9
                    recvcounts(*)
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
11
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12
         TYPE(*), DIMENSION(..) :: recvbuf
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     Fortran binding
17
     MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
18
                   RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
19
         <type> SENDBUF(*), RECVBUF(*)
20
         INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
21
                    RECVTYPE, COMM, IERROR
22
         This function supports Cartesian communicators, graph communicators, and distributed
23
     graph communicators as described in Section 8.6. If comm is a distributed graph commu-
24
     nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
25
     and receives from each of its incoming neighbors:
26
27
     MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
28
     int *srcs=(int*)malloc(indegree*sizeof(int));
29
     int *dsts=(int*)malloc(outdegree*sizeof(int));
30
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
31
                                outdegree, dsts, MPI_UNWEIGHTED);
32
     int k;
33
34
     /* assume sendbuf and recvbuf are of type (char*) */
35
     for(k=0; k<outdegree; ++k)</pre>
36
       MPI_Isend(sendbuf+sdispls[k]*extent(sendtype), sendcounts[k], sendtype,
37
                  dsts[k],...);
38
39
     for(k=0; k<indegree; ++k)</pre>
40
       MPI_Irecv(recvbuf+rdispls[k]*extent(recvtype), recvcounts[k], recvtype,
41
                  srcs[k],...);
42
43
     MPI_Waitall(...);
44
45
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process *i* must be equal to the type signature associated with recvcounts[l], recvtype with srcs[l]==iat process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data in the sendbuf beginning at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data received from the l-th incoming neighbor is placed into recvbuf beginning at offset rdispls[l] elements (in terms of the recvtype).

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

MPI\_NEIGHBOR\_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.

MPI\_NEIGHBOR\_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

			14
IN	sendbuf	starting address of send buffer (choice)	15
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor	16
			17
			18
IN	sdispls	integer array (of length outdegree). Entry j specifies	19 20
	Suispis	the displacement in bytes (relative to sendbuf) from	20 21
		which to take the outgoing data destined for	21
		neighbor j (array of integers)	22
IN	sendtypes	array of datatypes (of length outdegree). Entry j	24
IIN	senutypes	specifies the type of data to send to neighbor j (array of handles)	25
			26
	us sub uf	,	27
OUT	recvbuf	starting address of receive buffer (choice)	28
IN	recvcounts	non-negative integer array (of length indegree)	29
		specifying the number of elements that are received	30
		from each neighbor	31
IN	rdispls	integer array (of length indegree). Entry i specifies	32
		the displacement in bytes (relative to $recvbuf)$ at	33
		which to place the incoming data from neighbor <b>i</b>	34
		(array of integers)	35
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i	36
			37 38
		(array of handles)	38 39
IN	comm	communicator with topology structure (handle)	40
		$\mathbf{r} = \mathbf{G}_{\mathbf{r}}$	41
bindi	ng		42
	0	and wear drug and int and anota []	4.2

```
1
     int MPI_Neighbor_alltoallw_c(const void *sendbuf,
\mathbf{2}
                   const MPI_Count sendcounts[], const MPI_Aint sdispls[],
3
                   const MPI_Datatype sendtypes[], void *recvbuf,
4
                   const MPI_Count recvcounts[], const MPI_Aint rdispls[],
5
                   const MPI_Datatype recvtypes[], MPI_Comm comm)
6
     Fortran 2008 binding
7
     MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
8
                   recvcounts, rdispls, recvtypes, comm, ierror)
9
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
10
         INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
11
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
12
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
13
         TYPE(*), DIMENSION(...) :: recvbuf
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
18
                   recvcounts, rdispls, recvtypes, comm, ierror) !(_c)
19
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
21
                    recvcounts(*)
22
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
23
         TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
24
         TYPE(*), DIMENSION(...) :: recvbuf
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     Fortran binding
28
     MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
29
                   RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
30
         <type> SENDBUF(*), RECVBUF(*)
31
         INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
32
                    IERROR
33
         INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
34
35
         This function supports Cartesian communicators, graph communicators, and distributed
36
     graph communicators as described in Section 8.6. If comm is a distributed graph commu-
37
     nicator, the outcome is as if each process executed sends to each of its outgoing neighbors
38
     and receives from each of its incoming neighbors:
39
40
     MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
41
     int *srcs=(int*)malloc(indegree*sizeof(int));
42
     int *dsts=(int*)malloc(outdegree*sizeof(int));
43
     MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
44
                                outdegree, dsts, MPI_UNWEIGHTED);
45
     int k;
46
47
     /* assume sendbuf and recvbuf are of type (char*) */
48
     for(k=0; k<outdegree; ++k)</pre>
```

MPI\_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k], dsts[k],...);

```
for(k=0; k<indegree; ++k)
MPI_Irecv(recvbuf+rdispls[k], recvcounts[k], recvtypes[k], srcs[k],...);</pre>
```

MPI\_Waitall(...);

The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtypes[l] with srcs[l]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument **comm** must have identical values on all processes.

### 8.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 6.12.

#### 8.7.1 Nonblocking Neighborhood Gather

# MPI\_INEIGHBOR\_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

	comm, request)		28
IN	sendbuf	starting address of send buffer (choice)	29
IN	sendcount	number of elements sent to each neighbor	30
		(non-negative integer)	31
		(non nogative meeger)	32
IN	sendtype	datatype of send buffer elements (handle)	33
OUT	recvbuf	starting address of receive buffer (choice)	34
IN	recvcount	number of elements received from each neighbor	35
	recvcount	(non-negative integer)	36
		(non-negative integer)	37
IN	recvtype	datatype of receive buffer elements (handle)	38
IN	comm	communicator with topology structure (handle)	39
OUT	request	communication request (handle)	40
001	request	communication request (nanule)	41

# C binding

```
1
     int MPI_Ineighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,
\mathbf{2}
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
3
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
4
     Fortran 2008 binding
5
     MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
6
                   recvtype, comm, request, ierror)
7
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         INTEGER, INTENT(IN) :: sendcount, recvcount
9
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
16
                   recvtype, comm, request, ierror) !(_c)
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
19
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Request), INTENT(OUT) :: request
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     Fortran binding
25
     MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
26
                   RECVTYPE, COMM, REQUEST, IERROR)
27
         <type> SENDBUF(*), RECVBUF(*)
28
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
29
30
         This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER.
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_INEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request) <sup>2</sup>				
IN	sendbuf	starting address of send buffer (choice)	3	
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	4 5 6	
IN	sendtype	datatype of send buffer elements (handle)	7	
Ουτ	recvbuf	starting address of receive buffer (choice)	8	
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor	9 10 11 12	
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to <b>recvbuf</b> ) at which to place the incoming data from neighbor i	13 14 15	
IN	recvtype	datatype of receive buffer elements (handle)	16 17	
IN	comm	communicator with topology structure (handle)	18	
OUT	request	communication request (handle)	19 20	
<pre>int MPI_Ineighbor_allgatherv(const void *sendbuf, int sendcount,</pre>				
<pre>Fortran 2008 binding MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,</pre>				
<pre>MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request, ierror) !(_c) 44 TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbuf 45 INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount 46 TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype 47 TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf 48</pre>				

1 2 3 4 5 6 7	INTEG TYPE( TYPE( INTEG Fortran b	ER(KIND=MPI_ADDRESS_KIND MPI_Comm), INTENT(IN) :: MPI_Request), INTENT(OUT ER, OPTIONAL, INTENT(OUT)	) :: request
8 9 10 11 12	<type< td=""><td>DISPLS, RECVTYPE, CO &gt; SENDBUF(*), RECVBUF(*)</td><td></td></type<>	DISPLS, RECVTYPE, CO > SENDBUF(*), RECVBUF(*)	
13 14	This c	all starts a nonblocking varia	nt of MPI_NEIGHBOR_ALLGATHERV.
15 16 17	8.7.2 No	nblocking Neighborhood Allt	oall
18 19 20	MPI_INEIG	GHBOR_ALLTOALL(sendbuf, s comm, request)	endcount, sendtype, recvbuf, recvcount, recvtype,
21	IN	sendbuf	starting address of send buffer (choice)
22 23	IN	sendcount	number of elements sent to each neighbor (non-negative integer)
24 25	IN	sendtype	datatype of send buffer elements (handle)
26	OUT	recvbuf	starting address of receive buffer (choice)
27 28	IN	recvcount	number of elements received from each neighbor (non-negative integer)
29 30	IN	recvtype	datatype of receive buffer elements (handle)
31	IN	comm	communicator with topology structure (handle)
32 33	OUT	request	communication request (handle)
34 35 36 37 38	C binding int MPI_I	neighbor_alltoall(const MPI_Datatype sendtyp	void *sendbuf, int sendcount, e, void *recvbuf, int recvcount, e, MPI_Comm comm, MPI_Request *request)
39 40 41	int MPI_I	MPI_Datatype sendtyp	t void *sendbuf, MPI_Count sendcount, e, void *recvbuf, MPI_Count recvcount, e, MPI_Comm comm, MPI_Request *request)
42 43 44 45 46 47 48	MPI_Ineig TYPE( INTEG	recvtype, comm, requ	T(IN), ASYNCHRONOUS :: sendbuf unt, recvcount

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                       1
                                                                                       \mathbf{2}
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                       3
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       4
                                                                                       5
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                      6
              recvtype, comm, request, ierror) !(_c)
                                                                                      7
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                       8
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                      9
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                      10
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                      11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                      12
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                      13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      14
                                                                                      15
Fortran binding
                                                                                      16
MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
                                                                                      17
              RECVTYPE, COMM, REQUEST, IERROR)
                                                                                      18
    <type> SENDBUF(*), RECVBUF(*)
                                                                                      19
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
                                                                                      20
    This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.
                                                                                      21
                                                                                      22
                                                                                      23
                                                                                      ^{24}
                                                                                      25
                                                                                      26
                                                                                      27
                                                                                      28
                                                                                      29
                                                                                      30
                                                                                      ^{31}
                                                                                      32
                                                                                      33
                                                                                      34
                                                                                      35
                                                                                      36
                                                                                      37
                                                                                      38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
                                                                                      44
                                                                                      45
                                                                                      46
                                                                                      47
                                                                                      48
```

		(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts e, comm, request)
IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry j specifi the displacement (relative to sendbuf) from which send the outgoing data to neighbor j
IN	sendtype	datatype of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are receive from each neighbor
IN	rdispls	integer array (of length indegree). Entry i specifie the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
IN	recvtype	datatype of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
OUT	request	communication request (handle)
int MPI.	const int rec MPI_Datatype _Ineighbor_alltoall const MPI_Cou MPI_Datatype const MPI_Cou	<pre>ispls[], MPI_Datatype sendtype, void *recvbuf, cvcounts[], const int rdispls[], recvtype, MPI_Comm comm, MPI_Request *request) v_c(const void *sendbuf, unt sendcounts[], const MPI_Aint sdispls[], sendtype, void *recvbuf, unt recvcounts[], const MPI_Aint rdispls[], recvtype, MPI_Comm comm, MPI_Request *request)</pre>
Fortran	2008 binding	
MPI_Ine: TYPI	ighbor_alltoallv(se recvcounts, r E(*), DIMENSION()	<pre>endbuf, sendcounts, sdispls, sendtype, recvbuf, cdispls, recvtype, comm, request, ierror) , INTENT(IN), ASYNCHRONOUS :: sendbuf SYNCUPONOUS :: sendbuf</pre>
T N T 1		<pre>SYNCHRONOUS :: sendcounts(*), sdispls(*), ), rdispls(*)</pre>
TYP		TENT(IN) :: sendtype, recvtype
	E(*), DIMENSION()	, ASYNCHRONOUS :: recvbuf
TYPI		
TYPI TYPI	E(MPI_Comm), INTENT	
TYPI TYPI TYPI	E(MPI_Comm), INTENT	ENT(OUT) :: request

```
1
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
              recvcounts, rdispls, recvtype, comm, request, ierror) !(_c)
                                                                                    \mathbf{2}
                                                                                    3
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                    4
                                                                                    5
               sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                    6
                                                                                    7
               rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                    8
                                                                                    9
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    12
                                                                                    13
Fortran binding
                                                                                    14
MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
                                                                                    15
              RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
                                                                                    16
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    17
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                    18
               RECVTYPE, COMM, REQUEST, IERROR
                                                                                    19
                                                                                    20
    This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.
                                                                                    21
                                                                                    22
                                                                                    23
                                                                                    ^{24}
                                                                                    25
                                                                                    26
                                                                                    27
                                                                                    28
```

```
435
```

12	MPI_INEI	GHBOR_ALLTOALLW(sendbuf, rdispls, recvtypes, comm,	, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, request)
$\frac{3}{4}$	IN	sendbuf	starting address of send buffer (choice)
4 5 6 7	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
8 9 10 11	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
12 13 14 15	IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)
16	OUT	recvbuf	starting address of receive buffer (choice)
17 18 19	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
20 21 22 23 24	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
25 26 27	IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
28	IN	comm	communicator with topology structure (handle)
29 30	OUT	request	communication request (handle)
31 32 33 34 35 36 37	C binding int MPI_J	Ineighbor_alltoallw(const const MPI_Aint sdisp void *recvbuf, const	<pre>ls[], const MPI_Datatype recvtypes[],</pre>
38 39 40 41 42 43	int MPI_]	const MPI_Datatype s const MPI_Count recv	<pre>counts[], const MPI_Aint sdispls[], endtypes[], void *recvbuf, counts[], const MPI_Aint rdispls[], ecvtypes[], MPI_Comm comm,</pre>
$44 \\ 45$	Fortran 2	2008 binding	
46	MPI_Ineig	-	sendcounts, sdispls, sendtypes, recvbuf,
47 48	TYPE	-	recvtypes, comm, request, ierror) T(IN), ASYNCHRONOUS :: sendbuf

```
1
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                     2
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
               rdispls(*)
                                                                                     4
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                     5
               recvtypes(*)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                     6
                                                                                     7
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                     8
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     9
                                                                                    10
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                    11
              recvcounts, rdispls, recvtypes, comm, request, ierror) !(_c)
                                                                                    12
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    13
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                    14
               sendcounts(*), recvcounts(*)
                                                                                    15
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                    16
               rdispls(*)
                                                                                    17
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                    18
               recvtypes(*)
                                                                                    19
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    21
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    23
                                                                                    ^{24}
Fortran binding
                                                                                    25
MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
                                                                                    26
              RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
                                                                                    27
    <type> SENDBUF(*), RECVBUF(*)
                                                                                    28
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
                                                                                    29
               REQUEST, IERROR
                                                                                    30
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
                                                                                    31
    This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.
                                                                                    32
                                                                                    33
                                                                                    34
      Persistent Neighborhood Communication on Process Topologies
8.8
                                                                                    35
Persistent variants of the neighborhood collective operations can offer significant perfor-
                                                                                    36
mance benefits for programs with repetitive communication patterns. The semantics are
                                                                                    37
similar to persistent collective operations as described in Section 6.13.
                                                                                    38
                                                                                    39
                                                                                    40
```

```
438
                                                  CHAPTER 8. PROCESS TOPOLOGIES
1
     8.8.1 Persistent Neighborhood Gather
\mathbf{2}
3
4
     MPI_NEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount,
5
                    recvtype, comm, info, request)
6
       IN
                sendbuf
                                            starting address of send buffer (choice)
7
       IN
8
                sendcount
                                            number of elements sent to each neighbor
9
                                            (non-negative integer)
10
                                            datatype of send buffer elements (handle)
       IN
                sendtype
11
       OUT
                recvbuf
                                            starting address of receive buffer (choice)
12
       IN
                                            number of elements received from each neighbor
13
                 recvcount
14
                                            (non-negative integer)
15
       IN
                 recvtype
                                            datatype of receive buffer elements (handle)
16
       IN
                comm
                                            communicator with topology structure (handle)
17
       IN
                info
18
                                            info argument (handle)
19
       OUT
                request
                                            communication request (handle)
20
21
     C binding
22
     int MPI_Neighbor_allgather_init(const void *sendbuf, int sendcount,
23
                    MPI_Datatype sendtype, void *recvbuf, int recvcount,
24
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
25
                    MPI_Request *request)
26
27
     int MPI_Neighbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount,
                    MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
28
                    MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
29
30
                    MPI_Request *request)
^{31}
     Fortran 2008 binding
32
     MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
33
                    recvcount, recvtype, comm, info, request, ierror)
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
35
         INTEGER, INTENT(IN) :: sendcount, recvcount
36
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
37
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Info), INTENT(IN) :: info
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
44
                    recvcount, recvtype, comm, info, request, ierror) !(_c)
45
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
47
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
48
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

	E(MPI_Comm), INTENT(I		1
	E(MPI_Info), INTENT(I		2 3
	E(MPI_Request), INTEN EGER, OPTIONAL, INTEN	-	4
			5
	binding		6
MPI_NEI		SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, /TYPE, COMM, INFO, REQUEST, IERROR)	7
<tv< td=""><td>pe&gt; SENDBUF(*), RECVE</td><td></td><td>8</td></tv<>	pe> SENDBUF(*), RECVE		8
•	-	YPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	9 10 11
Cre operatio	-	e communication request for the neighborhood allgather	12 13 14
MPI_NE		INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, omm, info, request)	15 16 17
IN	sendbuf	starting address of send buffer (choice)	18
IN	sendcount	number of elements sent to each neighbor (non-negative integer)	19 20 21
IN	sendtype	datatype of send buffer elements (handle)	22
OUT	recvbuf	starting address of receive buffer (choice)	23
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor	24 25 26 27
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i	28 29 30
IN	recvtype	datatype of receive buffer elements (handle)	31
IN	comm	communicator with topology structure (handle)	32 33
IN	info	info argument (handle)	34
OUT	request	communication request (handle)	35 36
C bindi	ing		37
	0	init(const void *sendbuf, int sendcount,	38 39
	• •	endtype, void *recvbuf, const int recvcounts[],	40
<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>			41
		MPI_Request *request)	42
int MPI	<b>v</b>	init_c(const void *sendbuf,	43
		<pre>count, MPI_Datatype sendtype, void *recvbuf, t recvcounts[], const MPI_Aint displs[],</pre>	44 45
		ecvtype, MPI_Comm comm, MPI_Info info,	45 46
	MPI_Request *re		47
	_		48

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
3
                   recvcounts, displs, recvtype, comm, info, request, ierror)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER, INTENT(IN) :: sendcount, displs(*)
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
14
                   recvcounts, displs, recvtype, comm, info, request, ierror)
15
                   !(_c)
16
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
17
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
18
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
19
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         TYPE(MPI_Info), INTENT(IN) :: info
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     Fortran binding
28
     MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
29
                   RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
30
         <type> SENDBUF(*), RECVBUF(*)
31
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
32
                    INFO, REQUEST, IERROR
33
         Creates a persistent collective communication request for the neighborhood allgathery
34
     operation.
35
36
37
38
39
40
41
42
43
44
45
46
47
```

8.8.	PERSISTENT NEIGHBORHOOD COMMUNICATION	441
8.8.2	2 Persistent Neighborhood Alltoall	1 2
		3
MPI.	NEIGHBOR_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, r recvtype, comm, info, request)	ecvcount, 4 5
IN	sendbuf starting address of send buffer (cho	ice) <sup>6</sup>
IN		,
IN	sendtype datatype of send buffer elements (h	
οι	UT recvbuf starting address of receive buffer (cl	hoice) 11
IN	recvcount number of elements received from e (non-negative integer)	
IN	recvtype datatype of receive buffer elements	
IN	comm communicator with topology struct	ure (handle) <sup>16</sup> <sub>17</sub>
IN	info info argument (handle)	18
οι	UT request communication request (handle)	19
		20 21
	<pre>inding MPI_Neighbor_alltoall_init(const void *sendbuf, int sendco MPI_Datatype sendtype, void *recvbuf, int recvcou MPI_Datatype recvtype, MPI_Comm comm, MPI_Info in</pre>	22 Dunt, 23 unt, 24
	MPI_Request *request)	26
int	<pre>MPI_Neighbor_alltoall_init_c(const void *sendbuf, MPI_Coun MPI_Datatype sendtype, void *recvbuf, MPI_Count of MPI_Datatype recvtype, MPI_Comm comm, MPI_Info in MPI_Request *request)</pre>	recvcount, 28 nfo, 29 30
Fort	tran 2008 binding	31
MPI_	_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvb	ouf, 33
	recvcount, recvtype, comm, info, request, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbu	- 54
	INTEGER, INTENT(IN) :: sendcount, recvcount	30
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype	36 37
	<pre>TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf</pre>	38
	TYPE(MPI_Comm), INTENT(IN) :: comm	39
	TYPE(MPI_Info), INTENT(IN) :: info TYPE(MPI_Request), INTENT(OUT) :: request	40
	INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
мрт	_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvb	
· · · · · _	recvcount, recvtype, comm, info, request, ierror	Jul,
	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: sendbu	
	<pre>INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, rec</pre>	
	TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype TYPE(*), DIMENSION(), ASYNCHRONOUS :: recvbuf	47 48

1 2 3 4	TYPE TYPE	(MPI_Comm), INTENT(1 (MPI_Info), INTENT(1 (MPI_Request), INTEN GER, OPTIONAL, INTEN	IN) :: info NT(OUT) :: request
5 6 7 8 9 10 11	<typ< td=""><td>HBOR_ALLTOALL_INIT(S RECVCOUNT, RECV e&gt; SENDBUF(*), RECVE</td><td>SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, VTYPE, COMM, INFO, REQUEST, IERROR) SUF(*) TYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,</td></typ<>	HBOR_ALLTOALL_INIT(S RECVCOUNT, RECV e> SENDBUF(*), RECVE	SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, VTYPE, COMM, INFO, REQUEST, IERROR) SUF(*) TYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
12 13 14 15	Creat operation	_	ve communication request for the neighborhood all toall
16 17	MPI_NEI		IIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, s, recvtype, comm, info, request)
18	IN	sendbuf	starting address of send buffer (choice)
19 20 21 22	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
23 24 25	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j
26	IN	sendtype	datatype of send buffer elements (handle)
27 28	OUT	recvbuf	starting address of receive buffer (choice)
29 30 31	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
32 33 34	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
35 36	IN	recvtype	datatype of receive buffer elements (handle)
37	IN	comm	communicator with topology structure (handle)
38	IN	info	info argument (handle)
39 40	OUT	request	communication request (handle)
41 42	C bindin	ıg	
43	int MPI_	•	nit(const void *sendbuf,
44			counts[], const int sdispls[],
45 46		• -	<pre>endtype, void *recvbuf, const int recvcounts[], pls[], MPI_Datatype recvtype, MPI_Comm comm,</pre>
46 47			MPI_Request *request)
48			

```
1
int MPI_Neighbor_alltoallv_init_c(const void *sendbuf,
                                                                                   \mathbf{2}
              const MPI_Count sendcounts[], const MPI_Aint sdispls[],
                                                                                   3
              MPI_Datatype sendtype, void *recvbuf,
                                                                                   4
              const MPI_Count recvcounts[], const MPI_Aint rdispls[],
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
                                                                                   5
                                                                                   6
              MPI_Request *request)
                                                                                   7
Fortran 2008 binding
                                                                                   8
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
                                                                                   9
              recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
                                                                                   10
              ierror)
                                                                                   11
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   12
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                   13
              recvcounts(*), rdispls(*)
                                                                                   14
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   17
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   18
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
                                                                                   21
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
                                                                                   22
              recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
                                                                                   23
              ierror) !(_c)
                                                                                   24
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   25
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                   26
              sendcounts(*), recvcounts(*)
                                                                                   27
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   28
              rdispls(*)
                                                                                   29
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   30
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                   31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   32
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   35
Fortran binding
                                                                                   36
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
                                                                                   37
              RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
                                                                                   38
              IERROR)
                                                                                   39
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   40
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
                                                                                   41
              RECVTYPE, COMM, INFO, REQUEST, IERROR
                                                                                   42
                                                                                   43
    Creates a persistent collective communication request for the neighborhood alloally
                                                                                   44
operation.
                                                                                   45
                                                                                   46
```

12	MPI_NEIG		dbuf, sendcounts, sdispls, sendtypes, recvbuf, ypes, comm, info, request)
3	IN	sendbuf	starting address of send buffer (choice)
4 5 6 7	IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
8 9 10 11	IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
12 13 14 15	IN	sendtypes	array of data types (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)
16	OUT	recvbuf	starting address of receive buffer (choice)
17 18 19	IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
20 21 22 23 24	IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
25 26 27	IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
28	IN	comm	communicator with topology structure (handle)
29 30	IN	info	info argument (handle)
30 31	OUT	request	communication request (handle)
32 33 34 35 36 37 38 39 40 41 42 42	C binding int MPI_Neighbor_alltoallw_init(const void *sendbuf, const int sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const int recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info, MPI_Request *request) int MPI_Neighbor_alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[], const MPI_Aint sdispls[].		
$43 \\ 44 \\ 45 \\ 46 \\ 47$	Fortran 2	const MPI_Count recv	<pre>counts[], const MPI_Aint rdispls[], ecvtypes[], MPI_Comm comm, MPI_Info info,</pre>
48	MPI_Neigh	bor_alltoallw_init(sendbu	if, sendcounts, sdispls, sendtypes,

```
1
              recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                    2
              ierror)
                                                                                    3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    4
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                    5
                                                                                    6
               rdispls(*)
                                                                                    7
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                    8
               recvtypes(*)
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    9
                                                                                    10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    11
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                    12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                    13
                                                                                   14
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                    15
              recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                    16
              ierror) !(_c)
                                                                                    17
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                    18
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                    19
               sendcounts(*), recvcounts(*)
                                                                                   20
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   21
               rdispls(*)
                                                                                   22
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                   23
               recvtypes(*)
                                                                                    24
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                    25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                    26
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   29
                                                                                   30
Fortran binding
                                                                                    31
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
                                                                                   32
              RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
                                                                                   33
              IERROR)
                                                                                   34
    <type> SENDBUF(*), RECVBUF(*)
                                                                                   35
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
                                                                                   36
               INFO, REQUEST, IERROR
                                                                                   37
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
                                                                                   38
    Creates a persistent collective communication request for the neighborhood alltoallw
                                                                                    39
operation.
                                                                                    40
```

# 8.9 An Application Example

**Example 8.11** The example in Figures 8.4-8.7 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize

47 48

41 42

43 44

45

1 2 3 4 5 6 7 8 9	themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine relax. In each relaxation step each process computes new values for the solution grid function at the points $u(1:100,1:100)$ owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in $u(1,1:100)$ must be sent into the halo cells $u(101,1:100)$ of the left-hand neighbor with coordinates (own_coord(1)-1,own_coord(2)).
10	
11	
12 13	
14	
15	
16	
17	
18 19	
20	
21	
22	
23	
24	
25	
26 27	
27	
29	
30	
31	
32	
33	
34 35	
36	
37	
38	
39	
40	
41	
42 43	
43	
45	
46	
47	
48	

```
INTEGER ndims, num_neigh
                                                                                    1
LOGICAL reorder
                                                                                    \mathbf{2}
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
                                                                                    3
INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
                                                                                    4
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
                                                                                    5
LOGICAL periods(ndims)
                                                                                    6
REAL u(0:101,0:101), f(0:101,0:101)
                                                                                    7
DATA dims / ndims * 0 /
                                                                                    8
comm = MPI_COMM_WORLD
                                                                                    9
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
                                                                                    10
    Set process grid size and periodicity
!
                                                                                    11
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
                                                                                    12
periods(1) = .TRUE.
                                                                                    13
periods(2) = .TRUE.
                                                                                    14
    Create a grid structure in WORLD group and inquire about own position
1
                                                                                    15
CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
                                                                                    16
                      comm_cart, ierr)
                                                                                    17
CALL MPI_CART_GET(comm_cart, ndims, dims, periods, own_coords, ierr)
                                                                                    18
i = own_coords(1)
                                                                                    19
j = own_coords(2)
                                                                                    20
! Look up the ranks for the neighbors. Own process coordinates are (i,j).
                                                                                   21
! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
                                                                                   22
CALL MPI_CART_SHIFT(comm_cart, 0,1, neigh_rank(1), neigh_rank(2), ierr)
                                                                                   23
CALL MPI_CART_SHIFT(comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
                                                                                    24
! Initialize the grid functions and start the iteration
                                                                                    25
CALL init(u, f)
                                                                                    26
DO it=1,100
                                                                                    27
   CALL relax(u, f)
                                                                                    28
       Exchange data with neighbor processes
!
                                                                                    29
   CALL exchange(u, comm_cart, neigh_rank, num_neigh)
                                                                                    30
END DO
                                                                                    31
CALL output(u)
                                                                                    32
                                                                                    33
```

Figure 8.4: Set-up of process structure for two-dimensional parallel Poisson solver

34

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
1
     REAL u(0:101,0:101)
\mathbf{2}
     INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
3
     REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
4
     INTEGER ierr
5
     sndbuf(1:100,1) = u(1,1:100)
6
     sndbuf(1:100,2) = u(100,1:100)
7
     sndbuf(1:100,3) = u(1:100, 1)
8
     sndbuf(1:100,4) = u(1:100,100)
9
     CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
10
                                  comm_cart, ierr)
11
     ! instead of
12
     ! CALL MPI_IRECV(rcvbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(1), ierr)
13
     ! CALL MPI_ISEND(sndbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(2), ierr)
14
     !
         Always pairing a receive from rank_source with a send to rank_dest
15
         of the same direction in MPI_CART_SHIFT!
     !
16
     ! CALL MPI_IRECV(rcvbuf(1,2),100,MPI_REAL, neigh_rank(2),..., rq(3), ierr)
17
     ! CALL MPI_ISEND(sndbuf(1,1),100,MPI_REAL, neigh_rank(1),..., rq(4), ierr)
18
     ! CALL MPI_IRECV(rcvbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(5), ierr)
19
     ! CALL MPI_ISEND(sndbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(6), ierr)
20
     ! CALL MPI_IRECV(rcvbuf(1,4),100,MPI_REAL, neigh_rank(4),..., rq(7), ierr)
21
     ! CALL MPI_ISEND(sndbuf(1,3),100,MPI_REAL, neigh_rank(3),..., rq(8), ierr)
22
         Of course, one can first start all four IRECV and then all four ISEND,
     1
23
         Or vice versa, but both in the sequence shown above. Otherwise, the
     1
^{24}
     !
         matching would be wrong for 2 or only 1 processes in a direction.
25
     ! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)
26
     u(0,1:100) = rcvbuf(1:100,1)
27
     u(101,1:100) = rcvbuf(1:100,2)
28
     u(1:100, 0) = rcvbuf(1:100,3)
29
     u(1:100,101) = rcvbuf(1:100,4)
30
     END
^{31}
32
33
     Figure 8.5: Communication routine with local data copying and sparse neighborhood all-
34
     to-all
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
                                                                                         1
IMPLICIT NONE
                                                                                         2
USE MPI
                                                                                         3
REAL u(0:101,0:101)
                                                                                         4
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
                                                                                         5
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
                                                                                         6
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
INTEGER(KIND=MPI_ADDRESS_KIND) lb, sizeofreal
                                                                                         7
INTEGER(KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
                                                                                         8
INTEGER type_vec, ierr
                                                                                         9
! The following initialization need to be done only once
                                                                                         10
! before the first call of exchange.
                                                                                         11
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
                                                                                         12
CALL MPI_TYPE_VECTOR(100, 1, 102, MPI_REAL, type_vec, ierr)
                                                                                         13
CALL MPI_TYPE_COMMIT(type_vec, ierr)
sndtypes(1:2) = type_vec
                                                                                         14
sndcounts(1:2) = 1
                                                                                         15
sndtypes(3:4) = MPI_REAL
                                                                                         16
sndcounts(3:4) = 100
                                                                                         17
rcvtypes = sndtypes
                                                                                         18
rcvcounts = sndcounts
                                                                                         19
                                                                        , 1:100)
sdispls(1) = ( 1 + 1*102) * sizeofreal ! first element of u( 1
                                                                                        20
                    1*102) * sizeofreal ! first element of u(100
                                                                         1:100)
sdispls(2) = (100 +
                                                                                        21
sdispls(3) = (1 + 1*102) * size of real ! first element of u(1:100, 1)
                                                                                )
sdispls(4) = (1 + 100*102) * sizeofreal ! first element of u( 1:100,100
                                                                                )
                                                                                        22
rdispls(1) = (0 +
                     1*102) * sizeofreal ! first element of u( 0
                                                                      , 1:100)
                                                                                        23
                                                                        , 1:100)
rdispls(2) = (101 +
                      1*102) * sizeofreal ! first element of u(101
                                                                                        24
rdispls(3) = (1 +
                      0*102) * sizeofreal ! first element of u( 1:100, 0
                                                                                )
                                                                                        25
rdispls(4) = (1 + 101*102) * sizeofreal ! first element of u( 1:100,101
                                                                                )
                                                                                         26
! the following communication has to be done in each call of exchange
                                                                                        27
CALL MPI_NEIGHBOR_ALLTOALLW(u, sndcounts, sdispls, sndtypes, &
                                                                                        28
                            u, rcvcounts, rdispls, rcvtypes, &
                                                                                         29
                            comm_cart, ierr)
! The following finalizing need to be done only once
                                                                                         30
! after the last call of exchange.
                                                                                         31
CALL MPI_TYPE_FREE(type_vec, ierr)
                                                                                         32
END
                                                                                         33
                                                                                        34
                                                                                        35
Figure 8.6: Communication routine with sparse neighborhood all-to-all-w and without local
                                                                                        36
data copying
                                                                                        37
                                                                                         38
                                                                                         39
```

```
449
```

```
INTEGER ndims, num_neigh
1
     LOGICAL reorder
2
    PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
3
     INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
4
    LOGICAL periods(ndims)
5
    REAL u(0:101,0:101), f(0:101,0:101)
6
    DATA dims / ndims * 0 /
7
     INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
8
     INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
9
     INTEGER(KIND=MPI_ADDRESS_KIND) lb, sizeofreal
10
     INTEGER(KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
11
     INTEGER type_vec, request, status
12
     comm = MPI_COMM_WORLD
13
     CALL MPI_COMM_SIZE(comm, comm_size, ierr)
14
         Set process grid size and periodicity
15
    CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
16
     periods(1) = .TRUE.
17
    periods(2) = .TRUE.
18
         Create a grid structure in WORLD group
     !
19
     CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
20
                           comm_cart, ierr)
21
     ! Create datatypes for the neighborhood communication
22
     !
23
     ! Insert code from example in Figure 7.4 to create and initialize
24
     ! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
25
     Ţ
26
     ! Initialize the neighborhood all-to-all-w operation
27
     CALL MPI_NEIGHBOR_ALLTOALLW_INIT(u, sndcounts, sdispls, sndtypes, &
28
                                        u, rcvcounts, rdispls, rcvtypes, &
29
                                        comm_cart, info, request, ierr)
30
     ! Initialize the grid functions and start the iteration
31
     CALL init(u, f)
32
     DO it=1,100
33
            Start data exchange with neighbor processes
     !
34
        CALL MPI_START(request, ierr)
35
            Compute inner cells
     !
36
        CALL relax_inner (u, f)
37
            Check on completion of neighbor exchange
     !
38
        CALL MPI_WAIT(request, status, ierr)
39
            Compute edge cells
     1
40
        CALL relax_edges(u, f)
41
     END DO
42
     CALL output(u)
43
     CALL MPI_REQUEST_FREE(request, ierr)
44
     CALL MPI_TYPE_FREE(type_vec, ierr)
45
46
47
     Figure 8.7: Two-dimensional parallel Poisson solver with persistent sparse neighborhood
48
```

all-to-all-w and without local data copying

# Chapter 9

# **MPI** Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

 $^{24}$ 

#### Implementation Information 9.1

#### 9.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The "version" will be represented by two separate integers, for the version and subversion: In C,

#define MPI\_VERSION #define MPI\_SUBVERSION 0

in Fortran,				
INTEGER :: MPI_VERSION, MPI_S	UBVERSION	33		
PARAMETER (MPI_VERSION = 4	)	34		
PARAMETER (MPI_SUBVERSION = 0	)	35		
		36		
For runtime determination,		37		
		38		
MPI_GET_VERSION(version, subversion)	)	39		
		40		
OUT version	version number (integer)	41		
OUT subversion	subversion number (integer)	42		
		43		
C binding		44		
8	int MPI_Get_version(int *version, int *subversion)			
Fortran 2008 binding 47				
MPI_Get_version(version, subversion, ierror) 48				

12			:: version, subversion NTENT(OUT) :: ierror	
3	Fortran	binding		
4 5 6	MPI_GET_		SUBVERSION, IERROR) /ERSION, IERROR	
7				
8 9 10 11	always be	e thread-safe, as defin his and previous vers	be called at any time in an MPI program. This function must ned in Section 11.6. Valid (MPI_VERSION, MPI_SUBVERSION) sions of the MPI standard are $(4,0)$ , $(3,1)$ , $(3,0)$ , $(2,2)$ , $(2,1)$ ,	
11				
13	MPI_GET	LIBRARY_VERSIO	N(version, resultlen)	
14	OUT	version	version number (string)	
15 16	OUT	resultlen	Length (in printable characters) of the result returned in version (integer)	
17 18				
19	C bindir	ıg		
20	int MPI_	<u>Get_library_versi</u>	ion(char *version, int *resultlen)	
21	Fortran	2008 binding		
22		0	version, resultlen, ierror)	
23	CHAR	ACTER(LEN=MPI_MAX	<pre>LIBRARY_VERSION_STRING), INTENT(OUT) :: version</pre>	
24 25		INTEGER, INTENT(OUT) :: resultlen		
26	INTE	GER, OPTIONAL, IN	NTENT(OUT) :: ierror	
27	Fortran	binding		
28			/ERSION, RESULTLEN, IERROR)	
29		ACTER*(*) VERSION		
30 31		GER RESULTLEN, IE		
32 33			ring representing the version of the MPI library. The version g for maximum flexibility.	
34	A da	viao to implementare	. An implementation of MPI should return a different string	
35		-	ource code or build that could be visible to the user. ( <i>End of</i>	
36		ice to implementors.		
37		-	, ,	
38		0	ist represent storage that is	
39 40			_STRING characters long. MPI_GET_LIBRARY_VERSION may	
41	-	to this many charact	actually written is returned in the output argument, resultlen.	
42			onally stored at version[resultlen]. The value of resultlen cannot	
43	,		RARY_VERSION_STRING - 1. In Fortran, version is padded on	
44	-		ers. The value of resultlen cannot be larger than	
45		_LIBRARY_VERSION_		
46 47			RSION can be called at any time in an MPI program. This	
48	runction 1	nust always be threa	ad-safe, as defined in Section $11.6$ .	

9.1.2 Environmental Inquiries	1
When using the World Model (Section 11.2), a set of attributes that describe the execution environment is attached to the communicator MPI_COMM_WORLD when MPI is initialized.	2 3 4
The values of these attributes can be inquired by using the function MPI_COMM_GET_ATTR described in Section 7.7 and in Section 19.3.7. It is erroneous to	5 6
delete these attributes, free their keys, or change their values. The list of predefined attribute keys include	7 8
<b>MPI_TAG_UB</b> Upper bound for tag value.	9 10
<b>MPI_HOST</b> Host process rank, if such exists, MPI_PROC_NULL, otherwise.	11
<b>MPI_IO</b> rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.	12 13 14
<b>MPI_WTIME_IS_GLOBAL</b> Boolean variable that indicates whether clocks are synchronized.	15 16
When using the Sessions Model (Section 11.3), only the MPI_TAG_UB attribute is available.	10 17 18
Vendors may add implementation-specific parameters (such as node number, real mem- ory size, virtual memory size, etc.)	19 20
These predefined attributes do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.	21 22
Advice to users. Note that in the C binding, the value returned by these attributes is a <i>pointer</i> to an <b>int</b> containing the requested value. ( <i>End of advice to users.</i> )	23 24 25
The required parameter values are discussed in more detail below:	26 27
Tag Values	28 29
Tag values range from 0 to the value returned for MPI_TAG_UB, inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be <i>at least</i> 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a valid value for MPI_TAG_UB.	30 31 32 33 34
In the Sessions Model, the attribute MPI_TAG_UB is attached to all communicators created by MPI_COMM_CREATE_FROM_GROUP and MPI_INTERCOMM_CREATE_FROM_GROUPS, with the same value on all MPI processes in the communicator. In the World Model, the attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.	35 36 37 38 39
	40 41
Host Rank	42
The value returned for MPI_HOST gets the rank of the <i>HOST</i> process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if	43 44
there is no host. MPI does not specify what it means for a process to be a <i>HOST</i> , nor does	44
it requires that a $HOST$ exists.	46
The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.	47 48
	40

1	IO Rank					
2 3 4 5	I/O facilities (e.g., OPEN, I	. For Fortran, this means the REWIND, WRITE). For C, this	x of a processor that can provide language-standard nat all of the Fortran I/O operations are supported s means that all of the ISO C I/O operations are $\sim$			
6 7 8 9 10 11 12	<ul> <li>supported (e.g., fopen, fprintf, lseek).</li> <li>If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE</li> <li>will be returned. Otherwise, if the calling process can provide language-standard I/O</li> <li>then its rank will be returned. Otherwise, if some process can provide language-standard</li> <li>I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value</li> <li>MPI_PROC_NULL will be returned.</li> </ul>					
13 14 15		-	s not collective, and this attribute does <i>not</i> indicate input. ( <i>End of advice to users.</i> )			
16 17	Clock Synchr	onization				
<ol> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> <li>24</li> <li>25</li> <li>26</li> <li>27</li> <li>28</li> <li>29</li> <li>30</li> </ol>	The value returned for MPI_WTIME_IS_GLOBAL is 1 if clocks at all processes in MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to MPI_WTIME, will be less then one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one. The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This attribute may be associated with communicators other then MPI_COMM_WORLD. The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of MPI_COMM_WORLD.					
31 32 33	Inquire Proce	ssor Name				
34	MPI_GET_P	ROCESSOR_NAME(name, r	esultlen)			
35 36 37	OUT I	name	A unique specifier for the actual (as opposed to virtual) node.			
38 39 40	OUT I	resultlen	Length (in printable characters) of the result returned in $name$			
41 42 43	$\mathbf C$ binding int MPI_Get	_processor_name(char *1	name, int *resultlen)			
44	Fortran 200	08 binding				
45 46	<pre>MPI_Get_processor_name(name, resultlen, ierror)         CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name</pre>					
40	INTEGER, INTENT(OUT) :: resultlen					
48	INTEGEF	R, OPTIONAL, INTENT(OUT)	) :: ierror			

# Fortran binding MPI\_GET\_PROCESSOR\_NAME(NAME, RESULTLEN, IERROR) CHARACTER\*(\*) NAME INTEGER RESULTLEN, IERROR

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI\_MAX\_PROCESSOR\_NAME characters long. MPI\_GET\_PROCESSOR\_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI\_MAX\_PROCESSOR\_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI\_MAX\_PROCESSOR\_NAME.

*Rationale.* This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI\_GET\_PROCESSOR\_NAME simply allows such an implementation. (*End of rationale.*)

Advice to users. The user must provide at least MPI\_MAX\_PROCESSOR\_NAME space to write the processor name—processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

# 9.2 Memory Allocation

In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of some RMA functionality as defined in Section 12.5.3.

MPI\_ALLOC\_MEM(size, info, baseptr)

IN	size	size of memory segment in bytes (non-negative integer)	40 41
IN	info	info argument (handle)	42
OUT	baseptr	pointer to beginning of memory segment allocated	43
001		pointer to segmining of memory segment anotated	44
			45

C bindi	ng						
int MPI	_Alloc_mem(MP]	_Aint size	, MPI_Info	info,	void	*baseptr)	

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16 17

18

19

20

21 22

23

 $^{24}$ 

25

26 27

28 29

30

31

32

33

34

35

36 37 38

39

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Alloc_mem(size, info, baseptr, ierror)
3
          USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
4
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
5
          TYPE(MPI_Info), INTENT(IN) :: info
6
          TYPE(C_PTR), INTENT(OUT) :: baseptr
7
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     Fortran binding
9
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
10
          INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
11
          INTEGER INFO, IERROR
12
13
         If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must
14
     be provided in the mpi module and should be provided in mpif.h through overloading,
15
     i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND)
16
     BASEPTR, but with a different specific procedure name:
17
18
     INTERFACE MPI_ALLOC_MEM
19
          SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
              IMPORT :: MPI_ADDRESS_KIND
20
21
              INTEGER :: INFO, IERROR
22
              INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
23
          END SUBROUTINE
24
          SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
25
              USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
26
              IMPORT :: MPI_ADDRESS_KIND
27
              INTEGER :: INFO, IERROR
              INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
28
29
              TYPE(C_PTR) :: BASEPTR
30
          END SUBROUTINE
     END INTERFACE
31
32
         The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The
33
34
     implied specific procedure names are described in Section 19.1.5.
          By default, the allocated memory shall be aligned to at least the alignment required
35
     for load/store accesses of any datatype corresponding to a predefined MPI datatype. The
36
     info argument may be used to specify a desired alternative minimum alignment in bytes for
37
     the allocated memory by setting the value of the key "mpi_minimum_memory_alignment" to an
38
     integral number equal to a power of two. An implementation may ignore values smaller than
39
     the default required alignment. The info argument can also be used to provide directives
40
     that control the desired location of the allocated memory. Such a directive does not affect
41
     the semantics of the call. The corresponding info values are implementation-dependent. A
42
     null directive value of info = MPI_INFO_NULL is always valid.
43
         The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM
44
     to indicate it failed because memory is exhausted.
45
46
47
```

IN	base	initial address of memory segment allocated by
IIN	Dase	MPI_ALLOC_MEM (choice)
		······································
C bindi	ing	
	_Free_mem(void *bas	e)
ortran	2008 binding	
	e_mem(base, ierror)	
TYP	E(*), DIMENSION()	, INTENT(IN), ASYNCHRONOUS :: base
INT	EGER, OPTIONAL, INT	ENT(OUT) :: ierror
ortran	binding	
PI_FRE	E_MEM(BASE, IERROR)	
•	pe> BASE(*)	
INT	EGER IERROR	
		$MEM$ may return an error code of class $MPI\_ERR\_BASE$ to
ndicate	an invalid base argume	ent.
Ra	tionale. The C bindin	ngs of MPI_ALLOC_MEM and MPI_FREE_MEM are similar
	0	nalloc and free C library calls: a call to
		se) should be paired with a call to MPI_Free_mem(base) (one
	· · · · · · · · · · · · · · · · · · ·	Both arguments are declared to be of same type void* so
		ng. The Fortran binding is consistent with the C bindings: _MEM call returns in baseptr the TYPE(C_PTR) pointer or
		ress of the allocated memory. The base argument of
	,	oice argument, which passes (a reference to) the variable
$\operatorname{stc}$	ored at that location. (	End of rationale.)
Ad	lvice to implementors.	If MPI_ALLOC_MEM allocates special memory, then a
	-	gn of C malloc and free functions has to be used, in order
		memory segment, when the segment is freed. If no special
		LOC_MEM simply invokes malloc, and MPI_FREE_MEM
	okes free.	
		EM can be used in shared memory systems to allocate mem-
ory	y in a shared memory s	segment. (End of advice to implementors.)
-	me 4-byte REALs.	of MPI_ALLOC_MEM, in Fortran with TYPE(C_PTR) pointers.
	e e	
	pi_f08 ! or USE	
	INTRINSIC :: ISO_C_ (C_PTR) :: p	RINDING
	DIMENSION(:,:), PC	DINTER :: a ! no memory is allocated
	ER, DIMENSION(2) ::	•
	ER(KIND=MPI_ADDRESS	-
-	e = (/100, 100/)	
	= 4 * shape(1) * sh	

```
CALL MPI_Alloc_mem(size, MPI_INFO_NULL, p, ierr) ! memory is allocated and
  CALL C_F_POINTER(p, a, shape) ! intrinsic
                                                 ! now accessible via a(i,j)
                                   ! in ISO_C_BINDING
  . . .
  a(3,5) = 2.71
  . . .
  CALL MPI_Free_mem(a, ierr)
                                                     ! memory is freed
Example 9.2 Example of use of MPI_ALLOC_MEM, in Fortran with nonstandard Cray-
pointers. We assume 4-byte REALs, and assume that these pointers are address-sized.
  REAL A
  POINTER (P, A(100,100))
                              ! no memory is allocated
  INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
  SIZE = 4*100*100
  CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
  ! memory is allocated
  . . .
  A(3,5) = 2.71
  . . .
  CALL MPI_FREE_MEM(A, IERR) ! memory is freed
This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this code
or need a special option, e.g., the GNU gFortran compiler needs -fcray-pointer.
     Advice to implementors. Some compilers map Cray-pointers to address-sized integers,
     some to TYPE(C_PTR) pointers (e.g., Cray Fortran, version 7.3.3). From the user's
     viewpoint, this mapping is irrelevant because Examples 9.2 should work correctly
     with an MPI-3.0 (or later) library if Cray-pointers are available. (End of advice to
     implementors.)
Example 9.3 Same example, in C.
  float (* f)[100][100];
```

CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT

```
/* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */
...
(*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

# 9.3 Error Handling

458

1

2

3

4

5

6

7 8

9

10 11

12

13

14

15

16

17

18

19

20

21 22

23

24 25

26

27

28

29 30 31

32 33

34

35

36

37

38

39 40

41 42

43

<sup>44</sup> An MPI implementation may be unable or choose not to handle some failures that occur <sup>46</sup> during MPI calls. These can include failures that generate exceptions or traps, such as <sup>47</sup> floating point errors or access violations. The set of failures that are handled by MPI is <sup>48</sup> implementation-dependent. Each such failure causes an error to be raised. The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors *will* be handled should be read as may be handled. More background information about how MPI treats errors can be found in Section 2.8.

A user can associate error handlers to four types of objects: communicators, windows, files, and sessions. The specified error handling routine will be used for any error that occurs during a call to MPI for the respective object. MPI calls that are not related to any MPI objects are considered to be attached to the communicator MPI\_COMM\_SELF when using the World Model (see Section 11.2). When MPI\_COMM\_SELF is not initialized (i.e., before MPI\_INIT / MPI\_INIT\_THREAD, after MPI\_FINALIZE, or when using the Sessions Model exclusively) the error raises the initial error handler (set during the launch operation, see 11.8.4). The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

- **MPI\_ERRORS\_ARE\_FATAL** The handler, when called, causes the program to abort all connected MPI processes. This is similar to calling MPI\_ABORT using a communicator containing all connected processes with an implementation-specific value as the errorcode argument.
- **MPI\_ERRORS\_ABORT** The handler, when called, is invoked on a communicator in a manner similar to calling MPI\_ABORT on that communicator. If the error handler is invoked on an window or file, it is similar to calling MPI\_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. If the error handler is invoked on a session, the operation aborts only the local MPI process. In all cases, the value that would be provided as the errorcode argument to MPI\_ABORT is implementation-specific.
- **MPI\_ERRORS\_RETURN** The handler has no effect other than returning the error code to the user.

Advice to implementors. The implementation-specific error information resulting from MPI\_ERRORS\_ARE\_FATAL and MPI\_ERRORS\_ABORT provided to the invoking environment should be meaningful to the end-user, for example a predefined error class. (End of advice to implementors.)

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

Unless otherwise requested, the error handler MPI\_ERRORS\_ARE\_FATAL is set as the default initial error handler and associated with predefined communicators. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI\_ERRORS\_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a nontrivial MPI error handler. Note that unlike predefined communicators, windows and files do not inherit from the initial error handler, as defined in Sections 12.6 and 14.7 respectively. 

When an error is raised, MPI will provide the user information about that error using an error code. Some errors might prevent MPI from completing further API calls successfully 

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $^{31}$ 

and those functions will continue to report errors until the cause of the error is corrected
 or the user terminates the application. The user can make the determination of whether or
 not to attempt to continue when handling such an error.

Advice to users. For example, users may be unable to correct errors corresponding to some error classes, such as MPI\_ERR\_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors and available recovery actions. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C has distinct typedefs for user defined error handling callback functions that accept communicator, file, window, and session arguments. In Fortran there are four user routines.

An error handler object is created by a call to MPI\_XXX\_CREATE\_ERRHANDLER, where XXX is, respectively, COMM, WIN, FILE, or SESSION.

An error handler is attached to a communicator, window, file, or session by a call to MPI\_XXX\_SET\_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER, with matching XXX. An error handler can also be attached to a session using the errorhandler argument to MPI\_SESSION\_INIT. The predefined error handlers MPI\_ERRORS\_RETURN and MPI\_ERRORS\_ARE\_FATAL can be attached to communicators, windows, files, or sessions.

The error handler currently associated with a communicator, window, file, or session can be retrieved by a call to MPI\_XXX\_GET\_ERRHANDLER.

The MPI function MPI\_ERRHANDLER\_FREE can be used to free an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER.

MPI\_XXX\_GET\_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI\_ERRHANDLER\_FREE should be called with the error handler returned from MPI\_XXX\_GET\_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI\_COMM\_GROUP and MPI\_GROUP\_FREE.

36 37 38

39

40

41

42 43

32

33

34

35

- Advice to implementors. High-quality implementations should raise an error when an error handler that was created by a call to MPI\_XXX\_CREATE\_ERRHANDLER is attached to an object of the wrong type with a call to MPI\_YYY\_SET\_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)
- The syntax for these calls is given below.

4

5

6

7 8

9

10

11

12

9.3.1 Error	r Handlers for Communicato	rs	1		
			$^{2}$		
			3		
MPI_COMM	_CREATE_ERRHANDLER(co	omm_errhandler_fn, errhandler)	4		
	comm_errhandler_fn	,	5		
		user defined error handling procedure (function)	6		
OUT	errhandler	MPI error handler (handle)	7		
			8		
C binding	C binding				
int MPI_Com	mm_create_errhandler(		10		
	MPI_Comm_errhandler_1	function *comm_errhandler_fn,	11		
	MPI_Errhandler *errha	andler)	12		
Fortran 20	08 binding		13		
		rhandler_fn, errhandler, ierror)	14		
		function) :: comm_errhandler_fn	15 16		
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler			10		
	R, OPTIONAL, INTENT(OUT)		18		
<b>D</b> ( 1.			19		
Fortran binding					
		RHANDLER_FN, ERRHANDLER, IERROR)	20 21		
	AL COMM_ERRHANDLER_FN		22		
INTEGE	R ERRHANDLER, IERROR		23		
Creates	an error handler that can be	e attached to communicators.	24		
The use	er routine should be, in C, a	function of type MPI_Comm_errhandler_function,	25		
which is defi	ned as		26		
typedef vo:	id MPI_Comm_errhandler_f	unction(MPI_Comm *comm, int *error_code,	27		
	);		28		
The first argument is the communicator in use. The second is the error code to be					
	returned by the MPI routine that raised the error. If the routine would have returned				
e	MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused				
			32		

returned by the MPI routine that raised the error. If the routine would have returned MPI\_ERR\_IN\_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. With the Fortran mpi\_f08 module, the user routine comm\_errhandler\_fn should be of the form:

```
ABSTRACT INTERFACE
SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
TYPE(MPI_Comm) :: comm
INTEGER :: error_code
```

With the Fortran mpi module and mpif.h, the user routine COMM\_ERRHANDLER\_FN should be of the form:

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
```

INTEGER COMM, ERROR\_CODE

*Rationale.* The variable argument list is provided because it provides an ISOstandard hook for providing additional information to the error handler; without this

```
462
                                  CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT
1
          hook, ISO C prohibits additional arguments. (End of rationale.)
\mathbf{2}
3
          Advice to users.
                             A newly created communicator inherits the error handler that
          is associated with the "parent" communicator. In particular, the user can specify
4
          a "global" error handler for all communicators by associating this handler with the
5
          communicator MPI_COMM_WORLD immediately after initialization. (End of advice to
6
          users.)
7
8
9
10
     MPI_COMM_SET_ERRHANDLER(comm, errhandler)
11
       INOUT
                comm
                                            communicator (handle)
12
13
       IN
                errhandler
                                            new error handler for communicator (handle)
14
15
     C binding
16
     int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
17
     Fortran 2008 binding
18
19
     MPI_Comm_set_errhandler(comm, errhandler, ierror)
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     Fortran binding
^{24}
     MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
25
         INTEGER COMM, ERRHANDLER, IERROR
26
27
         Attaches a new error handler to a communicator. The error handler must be either
     a predefined error handler, or an error handler created by a call to
28
     MPI_COMM_CREATE_ERRHANDLER.
29
30
^{31}
     MPI_COMM_GET_ERRHANDLER(comm, errhandler)
32
33
       IN
                comm
                                            communicator (handle)
34
       OUT
                errhandler
                                            error handler currently associated with
35
                                            communicator (handle)
36
37
     C binding
38
     int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
39
40
     Fortran 2008 binding
41
     MPI_Comm_get_errhandler(comm, errhandler, ierror)
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     Fortran binding
46
     MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
47
         INTEGER COMM, ERRHANDLER, IERROR
48
```

9.3.2 Error Handlers for Windows

Retrieves the error handler currently associated with a communicator.

For example, a library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

			7			
			8			
MPI_WIN	I_CREATE_ERRHANDLER	(win_errhandler_fn, errhandler)	9 10			
IN	win_errhandler_fn	user defined error handling procedure (function)	11			
OUT	errhandler	MPI error handler (handle)	12			
001			13			
C bindi	ng		14			
	_Win_create_errhandler(	(	15			
	MPI_Win_errhandle	er_function *win_errhandler_fn,	16			
	MPI_Errhandler *	errhandler)	17 18			
Fortran	2008 binding		19			
	0	_errhandler_fn, errhandler, ierror)	20			
PROC	CEDURE(MPI_Win_errhand)	<pre>ler_function) :: win_errhandler_fn</pre>	21			
	E(MPI_Errhandler), INTE		22			
INTE	EGER, OPTIONAL, INTENT(	(OUT) :: ierror	23			
Fortran	binding		24			
MPI_WIN_	CREATE_ERRHANDLER(WIN_	ERRHANDLER_FN, ERRHANDLER, IERROR)	25 26			
	ERNAL WIN_ERRHANDLER_FN		20 27			
INTE	EGER ERRHANDLER, IERROF	ł	28			
Crea	tes an error handler that	can be attached to a window object. The user routine	29			
should be	e, in C, a function of type	MPI_Win_errhandler_function which is defined as	30			
typedef		er_function(MPI_Win *win, int *error_code,	31			
	);		32			
The	first argument is the win	ndow in use, the second is the error code to be re-	33			
	_	re "varargs" arguments whose number and meaning is	34 35			
-		elementation should clearly document these arguments.	36			
	- ,	the user routine win_errhandler_fn should be of the form:	37			
	INTERFACE	for the second	38			
	E(MPI_Win) :: win	er_function(win, error_code)	39			
	GER :: error_code		40			
	_		41			
	-	pif.h, the user routine WIN_ERRHANDLER_FN should	42 43			
	be of the form: SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)					
	GER WIN_ERROR_CODE	TION (WIN, ENGOIL_ODE)	44 45			
			46			
			47			
			48			

 $\mathbf{5}$ 

```
1
     MPI_WIN_SET_ERRHANDLER(win, errhandler)
\mathbf{2}
       INOUT
                win
                                            window object (handle)
3
       IN
                errhandler
                                            new error handler for window (handle)
4
5
6
     C binding
7
     int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
8
     Fortran 2008 binding
9
     MPI_Win_set_errhandler(win, errhandler, ierror)
10
          TYPE(MPI_Win), INTENT(IN) :: win
11
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
16
          INTEGER WIN, ERRHANDLER, IERROR
17
          Attaches a new error handler to a window. The error handler must be either a pre-
18
     defined error handler, or an error handler created by a call to
19
     MPI_WIN_CREATE_ERRHANDLER.
20
21
22
     MPI_WIN_GET_ERRHANDLER(win, errhandler)
23
       IN
                                            window object (handle)
                 win
^{24}
       OUT
                errhandler
                                            error handler currently associated with window
25
26
                                            (handle)
27
28
     C binding
29
     int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
30
     Fortran 2008 binding
^{31}
     MPI_Win_get_errhandler(win, errhandler, ierror)
32
          TYPE(MPI_Win), INTENT(IN) :: win
33
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
34
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     Fortran binding
37
     MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
38
          INTEGER WIN, ERRHANDLER, IERROR
39
         Retrieves the error handler currently associated with a window.
40
41
42
43
44
45
46
47
48
```

rror Handlers for Files		1
		2
		3
_CREATE_ERRHANDLER(fi	ile_errhandler_fn, errhandler)	4
•	,	5
		6
errhandler	MPI error handler (handle)	7
		8
ng		9
·		10
		11 12
MPI_Errhandler *er	rhandler)	12
2008 binding		13
	_errhandler_fn, errhandler, ierror)	15
EDURE(MPI_File_errhandle	er_function) :: file_errhandler_fn	16
(MPI_Errhandler), INTEN	T(OUT) :: errhandler	17
GER, OPTIONAL, INTENT(O	UT) :: ierror	18
Fortran binding		
		21
		22
		23
	-	24
		25
	r_function(MP1_File *file, int *error_code,	26
);		27
first argument is the file in us	se, the second is the error code to be returned. The re-	28
rguments are "varargs" arg	uments whose number and meaning is implementation-	29
t. An implementation should	d clearly document these arguments.	30 31
-	e user routine file_errhandler_fn should be of the form:	32
ABSTRACT INTERFACE		
SUBROUTINE MPI_File_errhandler_function(file, error_code)		
		35
GER :: error_code		36
Fortran mpi module and mpi	f.h, the user routine FILE_ERRHANDLER_FN should	37
form:		38
	TION(FILE, ERROR_CODE)	39
INTEGER FILE, ERROR_CODE		
		41
	E_CREATE_ERRHANDLER(fi file_errhandler_fn errhandler Pg File_create_errhandler( MPI_File_errhandler MPI_Errhandler *er 2008 binding _create_errhandler(file, EDURE(MPI_File_errhandle), INTEN GER, OPTIONAL, INTENT(O) binding E_CREATE_ERRHANDLER(FILE, RNAL FILE_ERRHANDLER(FILE, RNAL FILE_ERRHANDLER, IERROR tes an error handler that car a function of type MPI_File_ void MPI_File_errhandles ); first argument is the file in us rguments are "varargs" arg t. An implementation should Fortran mpi_f08 module, th INTERFACE TINE MPI_File_errhandles .(MPI_File) :: file GER :: error_code Fortran mpi module and mpi form: NE FILE_ERRHANDLER_FUNC	<pre>E_CREATE_ERRHANDLER(file_errhandler_fn, errhandler) file_errhandler_fn user defined error handling procedure (function) errhandler MPI error handler (handle)  gg File_create_errhandler(</pre>

```
1
     MPI_FILE_SET_ERRHANDLER(file, errhandler)
\mathbf{2}
       INOUT
                 file
                                             file (handle)
3
       IN
                 errhandler
                                             new error handler for file (handle)
4
5
6
     C binding
7
     int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
8
     Fortran 2008 binding
9
     MPI_File_set_errhandler(file, errhandler, ierror)
10
          TYPE(MPI_File), INTENT(IN) :: file
11
          TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
15
     MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
16
          INTEGER FILE, ERRHANDLER, IERROR
17
          Attaches a new error handler to a file. The error handler must be either a predefined
18
     error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
19
20
21
     MPI_FILE_GET_ERRHANDLER(file, errhandler)
22
       IN
                 file
                                             file (handle)
23
       OUT
                 errhandler
                                             error handler currently associated with file (handle)
^{24}
25
26
     C binding
27
     int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
28
     Fortran 2008 binding
29
     MPI_File_get_errhandler(file, errhandler, ierror)
30
          TYPE(MPI_File), INTENT(IN) :: file
^{31}
          TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     Fortran binding
35
     MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
36
          INTEGER FILE, ERRHANDLER, IERROR
37
         Retrieves the error handler currently associated with a file.
38
39
     9.3.4 Error Handlers for Sessions
40
41
42
43
     MPI_SESSION_CREATE_ERRHANDLER(session_errhandler_fn, errhandler)
44
       IN
                 session_errhandler_fn
                                             user defined error handling procedure (function)
45
       OUT
                 errhandler
                                             MPI error handler (handle)
46
47
48
```

C bindi	ng		1
	_Session_create_errl	handler(	2
	MPI_Session_e	rrhandler_function *session_errhandler_fn, r *errhandler)	3 4
<b>D</b> a	2008 1		5
	2008 binding	lon(accorion annhandlan fra annhandlan iannan)	6
		<pre>ler(session_errhandler_fn, errhandler, ierror) errhandler_function) :: session_errhandler_fn</pre>	7
		INTENT(OUT) :: errhandler	8
	EGER, OPTIONAL, INT		9
			10
	binding		11
		LER (SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)	12 13
	ERNAL SESSION_ERRHAN		13
TNT	EGER ERRHANDLER, IEI	RRUR	14
Crea	ates an error handler th	hat can be attached to a session object. In C, the	16
session_e	rrhandler_fn argument s	bould be a function of type MPI_Session_errhandler_function,	17
which is	defined as		18
typedef	void MPI_Session_e	rrhandler_function(MPI_Session *session,	19
	int *error_co	de,);	20
The	first argument is the s	session in use, the second is the error code to be returned.	21
	0	varargs" arguments whose number and meaning is imple-	22
nentation-dependent. An implementation should clearly document these arguments.			23
With the Fortran mpi_f08 module, the session_errhandler_fn argument should be of the			
form:	-		25
ABSTRAC'	Γ INTERFACE		26
SUBRO	JTINE MPI_Session_e	rrhandler_function(session, error_code)	27
TYP	E(MPI_Session) :: se	ession	28
INT	EGER :: error_code		29
With the	e Fortran mpi module	and mpif.h, the SESSION_ERRHANDLER_FN argument	30 31
	e of the form:		31
		LER_FUNCTION(SESSION, ERROR_CODE)	33
	EGER SESSION, ERROR		34
			35
			36
MPI_SES	SION_SET_ERRHAND	LER(session, errhandler)	37
INOUT	session	session (handle)	38
			39
IN	errhandler	new error handler for session (handle)	40
			41
C bindi	0		42
int MPI		dler(MPI_Session session,	43
	MPI_Errhandle	r errhandler)	44
Fortran	2008 binding		45
MPI_Ses	sion_set_errhandler	(session, errhandler, ierror)	46
TYP	E(MPI_Session), INT	ENT(IN) :: session	47 48
			40

```
1
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     Fortran binding
4
     MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
5
          INTEGER SESSION, ERRHANDLER, IERROR
6
7
         Attaches a new error handler to a session. The error handler must be either a pre-
8
     defined error handler, or an error handler created by a call to
9
     MPI_SESSION_CREATE_ERRHANDLER.
10
11
     MPI_SESSION_GET_ERRHANDLER(session, errhandler)
12
13
       IN
                session
                                            session (handle)
14
       OUT
                errhandler
                                            error handler currently associated with session
15
                                            (handle)
16
17
     C binding
18
     int MPI_Session_get_errhandler(MPI_Session session,
19
                    MPI_Errhandler *errhandler)
20
21
     Fortran 2008 binding
22
     MPI_Session_get_errhandler(session, errhandler, ierror)
23
         TYPE(MPI_Session), INTENT(IN) :: session
^{24}
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     Fortran binding
27
     MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)
28
         INTEGER SESSION, ERRHANDLER, IERROR
29
30
         Retrieves the error handler currently associated with a session.
^{31}
32
     9.3.5 Freeing Errorhandlers and Retrieving Error Strings
33
34
35
     MPI_ERRHANDLER_FREE(errhandler)
36
37
       INOUT
                errhandler
                                            MPI error handler (handle)
38
39
     C binding
40
     int MPI_Errhandler_free(MPI_Errhandler *errhandler)
41
     Fortran 2008 binding
42
     MPI_Errhandler_free(errhandler, ierror)
43
         TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
48
```

CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT

INTEGER	ERRHANDLER,	IERROR
---------	-------------	--------

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI\_ERRHANDLER\_NULL. The error handler will be deallocated after all the objects associated with it (communicator, window, or file) have been deallocated.

### MPI\_ERROR\_STRING(errorcode, string, resultlen)

INTEGER, INTENT(OUT) :: resultlen

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

IN	errorcode	Error code returned by an $MPI$ routine
OUT	string	Text that corresponds to the $errorcode$
OUT	resultlen	Length (in printable characters) of the result returned in string

#### C binding

<pre>int MPI_Error_string(int errorcode, char *string, int *resultlen)</pre>
Fortran 2008 binding
MPI_Error_string(errorcode, string, resultlen, ierror)
INTEGER, INTENT(IN) :: errorcode
CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string

### Fortran binding

		-				
MPI_	_ERROR_ST	FRING(E	ERRORCODE,	STRING,	RESULTLEN,	IERROR)
	INTEGER	ERRORC	CODE, RESU	ULTLEN, I	ERROR	
	CHARACTE	ER*(*)	STRING			

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI\_MAX\_ERROR\_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

This function must always be thread-safe, as defined in Section 11.6. It is one of the few routines that may be called before MPI is initialized or after MPI is finalized.

*Rationale.* The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI\_ERROR\_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER\*(\*) can not be referenced in, for example, a PRINT statement. (*End of rationale.*)

### 9.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI\_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI\_ERROR\_STRING).

All MPI function calls shall return MPI\_SUCCESS if and only if the specification of that function has been fulfilled at the point of return. For multiple completion functions,

 $^{31}$ 

if the function returns MPI\_ERR\_IN\_STATUS, the error code in each status object shall be
 set to MPI\_SUCCESS if and only if the specification of the operation represented by the
 corresponding MPI\_Request has been fulfilled at the point of return.

<sup>4</sup> When an operation raises an error, it may not satisfy its specification (for example, a <sup>5</sup> synchronizing operation may not have synchronized) and the content of the output buffers, <sup>6</sup> targeted memory, or output parameters is undefined. However, a valid error code shall <sup>7</sup> always be set when an operation raises an error, whether in the return value, error field in <sup>8</sup> the status object, or element in an array of error codes.

<sup>9</sup> To make it possible for an application to interpret an error code, the routine
 <sup>10</sup> MPI\_ERROR\_CLASS converts any error code into one of a small set of standard error codes,
 <sup>11</sup> called *error classes*. Valid error classes are shown in Table 9.1 and Table 9.2.

<sup>12</sup> The error classes are a subset of the error codes: an MPI function may return an error <sup>13</sup> class number; and the function MPI\_ERROR\_STRING can be used to compute the error <sup>14</sup> string associated with an error class. The values defined for MPI error classes are valid MPI <sup>15</sup> error codes.

<sup>16</sup> The error codes satisfy,

17

18 19

20

21 22

23

24

25 26

28

42

43

```
0 = MPI_SUCCESS < MPI_ERR_... \le MPI_ERR_LASTCODE.
```

*Rationale.* The difference between MPI\_ERR\_UNKNOWN and MPI\_ERR\_OTHER is that MPI\_ERROR\_STRING can return useful information about MPI\_ERR\_OTHER.

Note that  $MPI_SUCCESS = 0$  is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (*End of rationale.*)

27 MDL EDDOD, CLASS(array and a

MPI\_ERROR\_CLASS(errorcode, errorclass)

IN 29errorcode Error code returned by an MPI routine 30 OUT errorclass Error class associated with errorcode  $^{31}$ 32 C binding 33 int MPI\_Error\_class(int errorcode, int \*errorclass) 3435 Fortran 2008 binding 36 MPI\_Error\_class(errorcode, errorclass, ierror) 37 INTEGER, INTENT(IN) :: errorcode 38 INTEGER, INTENT(OUT) :: errorclass 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 40 Fortran binding 41

MPI\_ERROR\_CLASS (ERRORCODE, ERRORCLASS, IERROR) INTEGER ERRORCODE, ERRORCLASS, IERROR

<sup>44</sup> The function MPI\_ERROR\_CLASS maps each standard error code (error class) onto
 <sup>45</sup> itself.

This function must always be thread-safe, as defined in Section 11.6. It is one of the
 few routines that may be called before MPI is initialized or after MPI is finalized.

MPI_SUCCESS	No error	1
MPI_ERR_ACCESS	Permission denied	2
MPI_ERR_AMODE	Error related to the <b>amode</b> passed to	3
	MPI_FILE_OPEN	4
MPI_ERR_ARG	Invalid argument of some other kind	5
MPI_ERR_ASSERT	Invalid assertion argument	6
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	7
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM	8
MPI_ERR_BUFFER	Invalid buffer pointer argument	9
MPI_ERR_COMM	Invalid communicator argument	10
MPI_ERR_CONVERSION	An error occurred in a user supplied data	11
	conversion function	12
MPI_ERR_COUNT	Invalid count argument	13
MPI_ERR_DIMS	Invalid dimension argument	14
MPI_ERR_DISP	Invalid displacement argument	15
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	16
	tered because a data representation identi-	17
	fier that was already defined was passed to	18
	MPI_REGISTER_DATAREP	19
MPI_ERR_FILE	Invalid file handle argument	20
MPI_ERR_FILE_EXISTS	File exists	21
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	22
	the file is currently open by some process	23
MPI_ERR_GROUP	Invalid group argument	24
MPI_ERR_INFO	Invalid info argument	25
MPI_ERR_INFO_KEY	Key longer than $MPI_MAX_INFO_KEY$	26
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE	27
MPI_ERR_INFO_VALUE	Value longer than $MPI_MAX_INFO_VAL$	28
MPI_ERR_IN_STATUS	Error code is in status	29
MPI_ERR_INTERN	Internal MPI (implementation) error	30
MPI_ERR_IO	Other I/O error	31
MPI_ERR_KEYVAL	Invalid keyval argument	32
MPI_ERR_LOCKTYPE	Invalid locktype argument	33
MPI_ERR_NAME	Invalid service name passed to	34
	MPI_LOOKUP_NAME	35
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory	36
	is exhausted	37
MPI_ERR_NO_SPACE	Not enough space	38
MPI_ERR_NO_SUCH_FILE	File does not exist	39
MPI_ERR_NOT_SAME	Collective argument not identical on all	40
	processes, or collective routines called in	41
	a different order by different processes	42
		43
		44

Table 9.1: Error classes (Part 1)

1	MPI_ERR_OP	Invalid operation argument
2	MPI_ERR_OTHER	Known error not in this list
3	MPI_ERR_PENDING	Pending request
4	MPI_ERR_PORT	Invalid port name passed to
5		MPI_COMM_CONNECT
6	MPI_ERR_PROC_ABORTED	Operation failed because a peer process has
7		aborted
8	MPI_ERR_QUOTA	Quota exceeded
9	MPI_ERR_RANK	Invalid rank argument
10	MPI_ERR_READ_ONLY	Read-only file or file system
11	MPI_ERR_REQUEST	Invalid request argument
12	MPI_ERR_RMA_ATTACH	Memory cannot be attached (e.g., because
13		of resource exhaustion)
14	MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
15	MPI_ERR_RMA_FLAVOR	Passed window has the wrong flavor for the
16		called function
17	MPI_ERR_RMA_RANGE	Target memory is not part of the win-
18		dow (in the case of a window created
19		with MPI_WIN_CREATE_DYNAMIC, tar-
20		get memory is not attached)
21	MPI_ERR_RMA_SHARED	Memory cannot be shared (e.g., some pro-
22		cess in the group of the specified commu-
23		nicator cannot expose shared memory)
24	MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls
25	MPI_ERR_ROOT	Invalid root argument
26	MPI_ERR_SERVICE	Invalid service name passed to
27		MPI_UNPUBLISH_NAME
28	MPI_ERR_SESSION	Invalid session argument
29	MPI_ERR_SIZE	Invalid size argument
30	MPI_ERR_SPAWN	Error in spawning processes
31	MPI_ERR_TAG	Invalid tag argument
32	MPI_ERR_TOPOLOGY	Invalid topology argument
33	MPI_ERR_TRUNCATE	Message truncated on receive
34	MPI_ERR_TYPE	Invalid datatype argument
35	MPI_ERR_UNKNOWN	Unknown error
36	MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to
37		MPI_FILE_SET_VIEW
38	MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on
39		a file which supports sequential access only
40	MPI_ERR_VALUE_TOO_LARGE	Value is too large to store
41	MPI_ERR_WIN	Invalid window argument
42	MPI_ERR_LASTCODE	Last error code
43		
44		
45	Table 9.2: Err	or classes (Part 2)
46		
47		
48		

#### 9.5 Error Classes, Error Codes, and Error Handlers

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 14. For this purpose, functions are needed to:

1. add a new error class to the ones an MPI implementation already knows.

- 2. associate error codes with this error class, so that MPI\_ERROR\_CLASS works.
- 3. associate strings with these error codes, so that MPI\_ERROR\_STRING works.
- 4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes or codes: it is not expected that an application will generate them in significant numbers.

MPI\_ADD\_ERROR\_CLASS(errorclass)

OUT	errorclass	value for the new error class (integer)	19
			20
C bindin	G		21
	8 Add_error_class(int *err	orclass)	22
1110 III 1_/	Rud_error_crass(int werr		23
Fortran 2	2008 binding		24
MPI_Add_	error_class(errorclass,	ierror)	25
INTE	GER, INTENT(OUT) :: erro	orclass	26
INTE	GER, OPTIONAL, INTENT(OU	T) :: ierror	27
<b>Fortran</b>	hinding		28
	ERROR_CLASS (ERRORCLASS,	TEBBUB)	29
	GER ERRORCLASS, IERROR		30
			31
Creat	es a new error class and retu	rns the value for it.	32

Creates a new error class and returns the value for it.

Rationale. To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (End of rationale.)

Advice to users. Since a call to MPI\_ADD\_ERROR\_CLASS is local, the same errorclass may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same errorclass on all of the processes. Getting the "same" error on multiple processes may not cause the same value of error code to be generated. (End of advice to users.)

The value of MPI\_ERR\_LASTCODE is a constant value and is not affected by new user-43 defined error codes and classes. Instead, a predefined attribute key MPI\_LASTUSEDCODE is 44 associated with MPI\_COMM\_WORLD. The attribute value corresponding to this key is the 45current maximum error class including the user-defined ones. This is a local value and may 46 be different on different processes. The value returned by this key is always greater than or 47equal to MPI\_ERR\_LASTCODE. 48

1  $\mathbf{2}$ 

3

4

56

> 7 8

9

10 11

1213

14

15

1617

18

33

34

35 36

37

38

39

40

```
1
           Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change
\mathbf{2}
           unless the user calls a function to explicitly add an error class/code. In a multithreaded
3
           environment, the user must take extra care in assuming this value has not changed.
4
           Note that error codes and error classes are not necessarily dense. A user may not
5
           assume that each error class below MPI_LASTUSEDCODE is valid. (End of advice to
6
           users.)
7
8
9
     MPI_ADD_ERROR_CODE(errorclass, errorcode)
10
11
       IN
                 errorclass
                                              error class (integer)
12
       OUT
                 errorcode
                                              new error code to be associated with errorclass
13
                                              (integer)
14
15
     C binding
16
     int MPI_Add_error_code(int errorclass, int *errorcode)
17
18
     Fortran 2008 binding
19
     MPI_Add_error_code(errorclass, errorcode, ierror)
20
          INTEGER, INTENT(IN) :: errorclass
21
          INTEGER, INTENT(OUT) :: errorcode
22
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     Fortran binding
24
     MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
25
          INTEGER ERRORCLASS, ERRORCODE, IERROR
26
27
          Creates new error code associated with errorclass and returns its value in errorcode.
28
           Rationale. To avoid conflicts with existing error codes and classes, the value of the
29
30
           new error code is set by the implementation and not by the user. (End of rationale.)
31
32
33
     MPI_ADD_ERROR_STRING(errorcode, string)
34
                 errorcode
       IN
                                              error code or class (integer)
35
36
       IN
                 string
                                              text corresponding to errorcode (string)
37
38
     C binding
39
     int MPI_Add_error_string(int errorcode, const char *string)
40
41
     Fortran 2008 binding
42
     MPI_Add_error_string(errorcode, string, ierror)
          INTEGER, INTENT(IN) :: errorcode
43
          CHARACTER(LEN=*), INTENT(IN) :: string
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
48
```

CHAPTER 9. MPI ENVIRONMENTAL MANAGEMENT

INTEGER ERRORCODE, IERROR				
CHARA	CHARACTER*(*) STRING			
Assoc	Associates an error string with an error code or class. The string must be no more			
than MPI_I	MAX_ERROR_STRING characte	ers long. The length of the string is as defined in the	4 5	
calling lang	guage. The length of the strin	g does not include the null terminator in C. Trailing	6	
blanks will	be stripped in Fortran. Callin	$\operatorname{mg}$ MPI_ADD_ERROR_STRING for an errorcode that	7	
already ha	s a string will replace the old	l string with the new string. It is erroneous to call	8	
MPI_ADD	_ERROR_STRING for an error	r code or class with a value $\leq$ MPI_ERR_LASTCODE.	9	
		when no string has been set, it will return a empty	10	
- (	spaces in Fortran, "" in C).		11	
		for creating and associating error handlers with	12	
communica	ators, files, windows, and sess	ions.	13	
			14	
MPI COM	M_CALL_ERRHANDLER(com	ım. errorcode)	15	
	Υ.	,	16	
IN	comm	communicator with error handler (handle)	17	
IN	errorcode	error code (integer)	18	
			19	
C binding			20 21	
int MPI_C	Comm_call_errhandler(MP1_	Comm comm, int errorcode)	21	
Fortron 2008 hinding			23	
<pre>MPI_Comm_call_errhandler(comm, errorcode, ierror)</pre>			24	
TYPE(MPI_Comm), INTENT(IN) :: comm			25	
	ER, INTENT(IN) :: errorc		26	
INTEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	27	
Fortran b	binding		28	
	MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)			
INTEGER COMM, ERRORCODE, IERROR			30	
			31	
This function invokes the error handler assigned to the communicator with the error and supplied. This function returns MPI SUCCESS in C and the same value in TEPPOP if			32 33	
	code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error			
handler re	e	(assuming the process is not aborted and the error	34 35	
			35 36	
	<i>-</i>		37	
MPI_WIN_	CALL_ERRHANDLER(win, er	rorcode)	38	
IN	win	window with error handler (handle)	39	
IN	errorcode	error code (integer)	40	
			41	
C binding	2		42	
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)		43		
			44	
	2008 binding	nanda iannan)	45	
<pre>MPI_Win_call_errhandler(win, errorcode, ierror)     TYPE(MPI_Win), INTENT(IN) :: win</pre>			46	
	ER, INTENT(IN) :: errorc		47 48	
INTEGER, INTENT(IN) efforcode 48				

1	INT	EGER, OPTIONAI	, INTENT(OUT) :: ierror
2 3	Fortran binding		
4 5	MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR) INTEGER WIN, ERRORCODE, IERROR		
6 7 8 9 10	This function invokes the error handler assigned to the window with the error code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).		
11 12 13 14 15	MP	vice to users. I_ERRORS_ARE_ vice to users.)	In contrast to communicators, the error handler ATAL is associated with a window when it is created. ( <i>End of</i>
16	MPI FIL	E CALL ERRHA	NDLER(fh, errorcode)
17 18	IN	fh	file with error handler (handle)
19	IN	errorcode	error code (integer)
20 21 22 23	C bindi int MPI	0	handler(MPI_File fh, int errorcode)
24 25 26 27 28	MPI_Fil TYPI INTI	E(MPI_File), [ EGER, INTENT(]	ler(fh, errorcode, ierror) NTENT(IN) :: fh N) :: errorcode , INTENT(OUT) :: ierror
29 30 31 32	MPI_FIL	binding E_CALL_ERRHANI EGER FH, ERROI	LER(FH, ERRORCODE, IERROR) CODE, IERROR
33 34 35	This fund	$\operatorname{ction}\operatorname{returns}MF$	the error handler assigned to the file with the error code supplied. _SUCCESS in C and the same value in IERROR if the error handler suming the process is not aborted and the error handler returns).
36 37 38 39 40		vice to users. T vice to users.)	he default error handler for files is $MPI\_ERRORS\_RETURN.$ (End of
40 41	MPI SEG	SION CALL FF	RHANDLER(session, errorcode)
42 43	IN	session	session with error handler (handle)
43	IN	errorcode	error code (integer)
45 46 47 48	C bindi	0	errhandler(MPI_Session session, int errorcode)

Fortran 2008 binding
MPI_Session_call_errhandler(session, errorcode, ierror)
TYPE(MPI_Session), INTENT(IN) :: session
INTEGER, INTENT(IN) :: errorcode
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)
INTEGER SESSION, ERRORCODE, IERROR

This function invokes the error handler assigned to the session with the error code supplied. This function returns MPI\_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users are warned that handlers should not be called recursively with MPI\_COMM\_CALL\_ERRHANDLER, MPI\_FILE\_CALL\_ERRHANDLER, MPI\_WIN\_CALL\_ERRHANDLER, or MPI\_SESSION\_CALL\_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI\_COMM\_CALL\_ERRHANDLER, MPI\_FILE\_CALL\_ERRHANDLER, MPI\_SESSION\_CALL\_ERRHANDLER, MPI\_WIN\_CALL\_ERRHANDLER, or MPI\_SESSION\_CALL\_ERRHANDLER, is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (*End of advice to users.*)

### 9.6 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high resolution timers. See also Section 2.6.4.

MPI\_WTIME()
C binding
double MPI\_Wtime(void)
Fortran 2008 binding
DOUBLE PRECISION MPI\_Wtime()
Fortran binding
DOUBLE PRECISION MPI\_WTIME()

MPI\_WTIME returns a floating-point number of seconds, representing elapsed wallclock time since some time in the past.

 $^{24}$ 

```
1
          The "time in the past" is guaranteed not to change during the life of the process.
\mathbf{2}
      The user is responsible for converting large numbers of seconds to other units if they are
3
      preferred.
4
          This function is portable (it returns seconds, not "ticks"), and it allows high-resolution.
\mathbf{5}
      One would use it like this:
6
\overline{7}
      {
          double starttime, endtime;
8
9
          starttime = MPI_Wtime();
10
           ... stuff to be timed
                                      . . .
11
          endtime
                     = MPI_Wtime();
          printf("That took %f seconds\n", endtime-starttime);
12
      }
13
14
          The times returned are local to the node that called them. There is no requirement
15
      that different nodes return "the same time." (But see also the discussion of
16
      MPI_WTIME_IS_GLOBAL in Section 9.1.2).
17
18
19
      MPI_WTICK()
20
21
      C binding
22
      double MPI_Wtick(void)
23
      Fortran 2008 binding
^{24}
     DOUBLE PRECISION MPI_Wtick()
25
26
      Fortran binding
27
     DOUBLE PRECISION MPI_WTICK()
28
          MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns,
29
      as a double precision value, the number of seconds between successive clock ticks. For
30
      example, if the clock is implemented by the hardware as a counter that is incremented
31
      every millisecond, the value returned by MPI_WTICK should be (10^{-3}).
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

## Chapter 10

## The Info Object

Many of the routines in MPI take an argument info. info is an opaque object with a handle of type MPI\_Info in C and Fortran with the mpi\_f08 module, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI.

 $^{24}$ 

 $^{31}$ 

Some info hints allow the MPI library to restrict its support for certain operations in order to improve performance or resource utilization. If an application provides such an info hint, it must be compatible with any changes in the behavior of the MPI library that are allowed by the info hint.

An implementation must support info objects as caches for arbitrary (key,value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI\_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI\_INFO\_GET\_NKEYS, MPI\_INFO\_GET\_NTHKEY, and MPI\_INFO\_GET\_STRING must retain all (key,value) pairs so that layered functionality can also use the Info object.

Keys have an implementation-defined maximum length of MPI\_MAX\_INFO\_KEY, which is at least 32 and at most 255. Values have an implementation-defined maximum length of MPI\_MAX\_INFO\_VAL. In Fortran, leading and trailing spaces are stripped from both. Returned values will never be larger than these maximum lengths. Both key and value are case sensitive.

Rationale. Keys have a maximum length because the set of known keys will always be finite and known to the implementation and because there is no reason for keys to be complex. The small maximum size allows applications to declare keys of size MPI\_MAX\_INFO\_KEY. The limitation on value sizes is so that an implementation is not forced to deal with arbitrarily long strings. (*End of rationale.*)

Advice to users. MPI\_MAX\_INFO\_VAL might be very large, so it might not be wise to declare a string of that size. (*End of advice to users.*)

When info is used as an argument to any MPI routine, it is interpreted before that routine returns, so that it may be read, modified, or freed immediately after return. Changes to an info object after return from a routine do not affect that interpretation. 48

1 When the descriptions refer to a key or value as being a boolean, an integer, or a list,  $\mathbf{2}$ they mean the string representation of these types. An implementation may define its own 3 rules for how info value strings are converted to other types, but to ensure portability, every 4 implementation must support the following representations. Valid values for a boolean must  $\mathbf{5}$ include the strings "true" and "false" (all lowercase). For integers, valid values must include 6 string representations of decimal values of integers that are within the range of a standard  $\overline{7}$ integer type in the program. (However it is possible that not every integer is a valid value 8 for a given key.) On positive numbers, + signs are optional. No space may appear between 9 a + or - sign and the leading digit of a number. For comma separated lists, the string 10 must contain valid elements separated by commas. Leading and trailing spaces are stripped  $^{11}$ automatically from the types of info values described above and for each element of a comma 12separated list. These rules apply to all info values of these types. Implementations are free 13to specify a different interpretation for values of other info keys. 1415MPI\_INFO\_CREATE(info) 1617OUT info info object created (handle) 18 19C binding 20int MPI\_Info\_create(MPI\_Info \*info) 21Fortran 2008 binding 22 MPI\_Info\_create(info, ierror) 23TYPE(MPI\_Info), INTENT(OUT) :: info  $^{24}$ INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2526Fortran binding 27MPI\_INFO\_CREATE(INFO, IERROR) 28INTEGER INFO, IERROR 29 MPI\_INFO\_CREATE creates a new info object. The newly created object contains no 30 key/value pairs.  $^{31}$ 32 33 MPI\_INFO\_SET(info, key, value) 34 INOUT info info object (handle) 35 36 IN key key (string) 37 IN value value (string) 38 39 C binding 40 int MPI\_Info\_set(MPI\_Info info, const char \*key, const char \*value) 41 42Fortran 2008 binding 43 MPI\_Info\_set(info, key, value, ierror) 44TYPE(MPI\_Info), INTENT(IN) :: info 45CHARACTER(LEN=\*), INTENT(IN) :: key, value 46 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4748

Fortran binding			1
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)			2 3
	INTEGER INFO, IERROR		
CHARA	ACTER*(*) KEY, VALUE		4 5
MPI_I	NFO_SET adds the (key,value	) pair to info, and overrides the value if a value for	6
the same k	tey was previously set. $key$ and	d value are null-terminated strings in C. In Fortran,	7
0		value are stripped. If either key or value are larger	8
	*	MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are	9
raised, resp	pectively.		10
			11
MPI_INFO	_DELETE(info, key)		12
INOUT	info	info object (bandle)	13
		info object (handle)	14
IN	key	key (string)	15
			16
C binding			17 18
int MPI_I	nfo_delete(MPI_Info info	, const char *key)	19
Fortran 2	2008 binding		20
MPI_Info_	delete(info, key, ierror	)	21
	<pre>MPI_Info), INTENT(IN) ::</pre>		22
	CTER(LEN=*), INTENT(IN)		23
INTEG	ER, OPTIONAL, INTENT(OUT	) :: ierror	24
Fortran b	binding		25
MPI_INFO_DELETE(INFO, KEY, IERROR)			26
INTEGER INFO, IERROR			27
CHARACTER*(*) KEY			28
MPI I	NFO DELETE deletes a (key.	value) pair from info. If key is not defined in info,	29 30
	ises an error of class MPI_ERR		31
	_		32
			33
MPI_INFO	_GET_STRING(info, key, bufle	en, value, flag)	34
IN	info	info object (handle)	35
IN	key	key (string)	36
INOUT	buflen	length of buffer (integer)	37
			38
OUT	value	value (string)	39
OUT	flag	true if key defined, false if not (logical)	40 41
			41
C binding			43
<pre>int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen,</pre>			44
char *value, int *flag)			45
Fortran 2008 binding			46
			47
TYPE(MPI_Info), INTENT(IN) :: info48			48

482	

1 2 3 4 5 6	CHARACTER(LEN=*), INTENT(IN) :: key INTEGER, INTENT(INOUT) :: buflen CHARACTER(LEN=*), INTENT(OUT) :: value LOGICAL, INTENT(OUT) :: flag INTEGER, OPTIONAL, INTENT(OUT) :: ierror			
7 8 9 10 11	Fortran binding MPI_INFO_GET_STRING(INFO, KEY, BUFLEN, VALUE, FLAG, IERROR) INTEGER INFO, BUFLEN, IERROR CHARACTER*(*) KEY, VALUE LOGICAL FLAG			
12 13 14 15 16 17 18 19 20 21 22	This function retrieves the value associated with key from info, if any. If such a key exists in info, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. buflen on input is the size of the provided buffer, for the output of buflen it is the size of the buffer needed to store the value string. If the buflen passed into the function is less than the actual size needed to store the value string (including null terminator in C), the value is truncated. On return, the value of buflen will be set to the required buffer size to hold the value string. If buflen is set to 0, value is not changed. In C, buflen includes the required space for the null terminator. In C, this function returns a null terminated string in all cases where the buflen input value is greater than 0. If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.			
23 24 25 26 27 28	Advice to users. The MPI_INFO_GET_STRING function can be used to obtain the size of the required buffer for a value string by setting the buflen to 0. The returned buflen can then be used to allocate memory before calling MPI_INFO_GET_STRING again to obtain the value string. ( <i>End of advice to users.</i> )			
29 30	MPI_INFO	D_GET_NKEY	S(info, nkeys)	
31	IN	info	info object (handle)	
32 33	OUT	nkeys	number of defined keys (integer)	
34 35 36	C binding int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)			
37 38 39 40 41	<pre>Fortran 2008 binding MPI_Info_get_nkeys(info, nkeys, ierror)     TYPE(MPI_Info), INTENT(IN) :: info     INTEGER, INTENT(OUT) :: nkeys     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			
42 43 44 45	Fortran binding MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR) INTEGER INFO, NKEYS, IERROR			
46 47 48	MPI_	INFO_GET_N	<b>KEYS</b> returns the number of currently defined keys in info.	

			1
MPI_INFO_GET_NTHKEY(info, n, key)			
IN	info	info object (handle)	2 3
IN	n	key number (integer)	4
OUT	key	key (string)	5
	5		6
C bindin	g		7
int MPI_	Info_get_nthkey(MPI_Info	info, int n, char *key)	8
Fortran (	2008 binding		9 10
	_get_nthkey(info, n, key,	ierror)	10
	(MPI_Info), INTENT(IN) ::		12
INTE	GER, INTENT(IN) :: n		13
CHAR	ACTER(LEN=*), INTENT(OUT)	:: key	14
INTE	GER, OPTIONAL, INTENT(OUT	') :: ierror	15
<b>Fortran</b>	binding		16
	_GET_NTHKEY(INFO, N, KEY,	IERROR)	17
INTE	GER INFO, N, IERROR		18 19
CHAR	ACTER*(*) KEY		20
This	function returns the nth define	ed key in info. Keys are numbered $0 \dots N - 1$ where	21
		_GET_NKEYS. All keys between 0 and $N-1$ are	22
guarantee	d to be defined. The number	of a given key does not change as long as info is not	23
modified v	with MPI_INFO_SET or MPI_	INFO_DELETE.	24
			25
MPI_INFC	_DUP(info, newinfo)		26 27
IN	info	info object (handle)	27
	newinfo		29
OUT	newinio	info object created (handle)	30
C bindin	a		31
	g Info_dup(MPI_Info info, M	IPT Info *newinfo)	32
	-		33
	2008 binding	、 、	34
	_dup(info, newinfo, ierro		35 36
	(MPI_Info), INTENT(IN) :: (MPI_Info), INTENT(OUT) :		37
	GER, OPTIONAL, INTENT(OUT		38
			39
Fortran	0	ר תו	40
	MPI_INFO_DUP(INFO, NEWINFO, IERROR) INTEGER INFO, NEWINFO, IERROR		
			42 43
	MPI_INFO_DUP duplicates an existing info object, creating a new object, with the		
same (key, value) pairs and the same ordering of keys.			44 45
			45 46

```
1
     MPI_INFO_FREE(info)
\mathbf{2}
       INOUT
                 info
                                              info object (handle)
3
4
      C binding
\mathbf{5}
      int MPI_Info_free(MPI_Info *info)
6
7
     Fortran 2008 binding
8
     MPI_Info_free(info, ierror)
9
          TYPE(MPI_Info), INTENT(INOUT) :: info
10
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     Fortran binding
12
     MPI_INFO_FREE(INFO, IERROR)
13
          INTEGER INFO, IERROR
14
15
          This function frees info and sets it to MPI_INFO_NULL.
16
17
      MPI_INFO_CREATE_ENV(info)
18
19
       OUT
                 info
                                              info object (handle)
20
21
      C binding
22
      int MPI_Info_create_env(int argc, char argv[], MPI_Info *info)
23
24
     Fortran 2008 binding
25
     MPI_Info_create_env(info, ierror)
26
          TYPE(MPI_Info), INTENT(OUT) :: info
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
      Fortran binding
29
     MPI_INFO_CREATE_ENV(INFO, IERROR)
30
          INTEGER INFO, IERROR
^{31}
32
          This routine produces an output object info with the same construction as
33
      MPI_INFO_ENV as created during MPI_INIT or MPI_INIT_THREAD when the same argu-
34
      ments are used. This construction is described in Section 11.2.1; however, this function can
35
      be called when not using the World Model, e.g., when using the Sessions Model. This object
36
      is not a direct copy or alias of the MPI_INFO_ENV object and could contain different values
37
      based on the input arguments and other sources. Multiple calls to this procedure that are
38
      given the same input arguments will produce info objects consistent with the definition of
39
      MPI_INFO_ENV. The version for ISO C accepts the argc and argv that are provided by the
40
      arguments to main or 0 for argc and NULL for argv. The user is responsible for freeing the
^{41}
      info object via MPI_INFO_FREE. This procedure is local.
42
          This procedure must always be thread-safe, as defined in Section 11.6. It is one of the
43
      few routines that may be called before MPI is initialized or after MPI is finalized.
44
           Advice to users.
45
46
           In some circumstances (e.g., when passing 0 to argc and NULL to argv in C or in Fortran
47
           where such arguments do not exist), the info object may not be populated or may be
```

<sup>48</sup> populated incompletely because this procedure is local and the implementation may

not be able to determine the correct values. Note that this could result in different values in the resulting info object at different MPI processes.	
End of advice to users.)	

## Chapter 11

# Process Initialization, Creation, and Management

### 11.1 Introduction

MPI is primarily concerned with communication rather than process or resource management. However, it is necessary to address these issues to some degree in order to define a useful framework for communication. This chapter presents a set of MPI interfaces that allows for several approaches to MPI initialization and process management while placing minimal restrictions on the execution environment.

 $^{24}$ 

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup or initialization procedure to be performed before the complete set of MPI routines may be called.

To this end, MPI presents two models for **MPI process initialization**. In the World Model, an initial set of processes is created that are related by their membership in a common MPI\_COMM\_WORLD (see Section 11.2) communicator. In the Sessions Model (Section 11.3), an initial set of processes is also created, but the application must explicitly manage the creation of MPI groups, and hence MPI communicators. MPI\_COMM\_WORLD is only valid for use as a communicator in the World Model, i.e., after a successful call to MPI\_INIT\_THREAD and before a call to MPI\_FINALIZE. An application can employ both of these Process Models concurrently. In multi-component MPI applications, for example, a component such as a library can make use of the Sessions Model to instantiate MPI resources without impacting the rest of the application.

Both of these models also support the Dynamic Process Model (see Section 11.7), which provides for the creation and management of additional processes after an MPI application has been started. A major impetus for the Dynamic Process Model comes from the PVM [25] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

In developing the Dynamic Process Model, the MPI Forum decided not to address resource control because it was not able to design a portable interface that would be ap-

### 488 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

propriate for the broad spectrum of existing and potential resource and process controllers.
 MPI assumes that resource control is provided externally.

<sup>3</sup> Process management functionality is included in MPI to enable its use in classes of <sup>4</sup> message-passing applications requiring process control. These include task farms, serial <sup>5</sup> applications with parallel modules, and problems that require a run-time assessment of the <sup>6</sup> number and type of processes that should be started.

The following goals are central to the design of MPI process management:

- The MPI process model must apply to the vast majority of current parallel environments.
- MPI must not take over operating system responsibilities. It should instead provide a clean interface between an application and system software.
- MPI must guarantee communication determinism in the presence of dynamic processes, i.e., dynamic process management must not introduce unavoidable race conditions.
  - MPI must not contain features that compromise performance.

The Dynamic Process Model addresses these issues in two ways. First, MPI remains primarily a communication library. It does not manage the parallel environment in which a parallel program executes, though it provides a minimal interface between an application and external resource and process managers.

Second, MPI maintains a consistent concept of a communicator, regardless of how its
 members came into existence. A communicator is never changed once created, and it is
 always created using deterministic collective operations.

### <sup>28</sup> 11.2 The World Model

7

8

9

10 11

12

13 14

15

16

17

18 19

27

32

33

<sup>30</sup> 11.2.1 Starting MPI Processes <sup>31</sup>

When using the World Model, MPI is initialized by calling either MPI\_INIT or MPI\_INIT\_THREAD.

34 35 MPI\_INIT() 36 37 C binding 38 int MPI\_Init(int \*argc, char \*\*\*argv) 39 Fortran 2008 binding 40 MPI\_Init(ierror) 41 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4243 Fortran binding 44MPI\_INIT(IERROR) 45INTEGER IERROR 46 In the World Model, an MPI program must contain exactly one call to an MPI ini-47tialization routine: MPI\_INIT or MPI\_INIT\_THREAD. MPI\_COMM\_WORLD and 48

MPI\_COMM\_SELF are not valid for use as communicators prior to invocation of MPI\_INIT or MPI\_INIT\_THREAD. Subsequent calls to either of these initialization routines are erroneous. A subset of MPI functions may be invoked before MPI initialization routines are called. See Section 11.4. MPI\_INIT accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    /* parse arguments */
    /* main program */
    MPI_Finalize();    /* see below */
    return 0;
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C.

Failures may disrupt the execution of the program before or during MPI initialization. A high-quality implementation shall not deadlock during MPI initialization, even in the presence of failures. Except for functions with the MPI\_T\_ prefix, failures in MPI operations prior to or during MPI initialization are reported by invoking the initial error handler. Users can use the "mpi\_initial\_errhandler" info key during the launch of MPI processes (e.g., MPI\_COMM\_SPAWN / MPI\_COMM\_SPAWN\_MULTIPLE, or mpiexec) to set a non-fatal initial error handler before MPI initialization. When the initial error handler is set to MPI\_ERRORS\_ABORT, raising an error before or during initialization aborts the local MPI process (i.e., it is similar to calling MPI\_ABORT on MPI\_COMM\_SELF). An implementation may not always be capable of determining, before MPI initialization, what constitutes the local MPI process, or the set of connected processes. In this case, errors before initialization, the initial error handler is associated with MPI\_COMM\_WORLD, MPI\_COMM\_SELF, and the communicator returned by MPI\_COMM\_GET\_PARENT (if any).

Advice to implementors. Some failures may leave MPI in an undefined state, or raise an error before the error handling capabilities are fully operational, in which cases the implementation may be incapable of providing the desired error handling behavior. Of note, in some implementations, the notion of an MPI process is not clearly established in the early stages of MPI initialization (for example, when the implementation considers threads that called MPI\_INIT as independent MPI processes); in this case, before MPI is initialized, the MPI\_ERRORS\_ABORT error handler may abort what would have become multiple MPI processes.

When a failure occurs during MPI initialization, the implementation may decide to return MPI\_SUCCESS from the MPI initialization function instead of raising an error. It is recommended that an implementation masks an initialization error only when it expects that later MPI calls will result in well-specified behavior (i.e., barring additional failures, either the outcome of any call will be correct, or the call will raise an 

 $\mathbf{2}$ 

 $^{24}$ 

### 490 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

appropriate error). For example, it may be difficult for an implementation to avoid unspecified behavior when the group of MPI\_COMM\_WORLD does not contain the same set of MPI processes at all members of the communicator, or if the communicator returned from MPI\_COMM\_GET\_PARENT was not initialized correctly. (End of advice to implementors.)

After MPI is initialized, the application can access information about the execution  $\overline{7}$ environment by querying the predefined info object MPI\_INFO\_ENV. The following keys are 8 predefined for this object, corresponding to the arguments of MPI\_COMM\_SPAWN or of 9 mpiexec: 10

- 11"command" Name of program executed. 12
- "argv" Space separated arguments to command. 13
- 14"maxprocs" Maximum number of MPI processes to start. 15
- 16"mpi\_initial\_errhandler" Name of the initial errhandler. 17
- "soft" Allowed values for number of processors. 18
- 19"host" Hostname. 20

1

 $\mathbf{2}$ 

3

4

5

6

22

28

29

30

 $^{31}$ 32

33

34

35

36 37

38

39

 $^{41}$ 

43

4445

47

48

- 21"arch" Architecture name.
- "wdir" Working directory of the MPI process. 23
- $^{24}$ "file" Value is the name of a file in which additional information is specified. 25
- 26"thread\_level" Requested level of thread support, if requested before the program started execution. 27

Note that all values are strings. Thus, the maximum number of processes is represented by a string such as "1024" and the requested level is represented by a string such as "MPI\_THREAD\_SINGLE".

Advice to users. If one of the "argv" arguments contains a space, there is no way to tell from the value of the "argv" info key whether a space is part of the argument or is separating different arguments. (End of advice to users.)

The info object MPI\_INFO\_ENV need not contain a (key,value) pair for each of these predefined keys; the set of (key, value) pairs provided is implementation-dependent. Implementations may provide additional, implementation specific, (key, value) pairs.

In cases where the MPI processes were started with MPI\_COMM\_SPAWN\_MULTIPLE or, equivalently, with a startup mechanism that supports multiple process specifications, 40then the values stored in the info object MPI\_INFO\_ENV at a process are those values that affect the local MPI process. 42

```
Example 11.1 If MPI is started with a call to
          mpiexec -n 5 -arch x86_64 ocean : -n 10 -arch power9 atmos
46
     Then the first 5 processes will have in their MPI_INFO_ENV object the pairs (command,
     ocean), (maxprocs, 5), and (arch, x86_64). The next 10 processes will have in MPI_INFO_ENV
```

(command, atmos), (maxprocs, 10), and (arch, power9)

Advice to users. The values passed in MPI\_INFO\_ENV are the values of the arguments passed to the mechanism that started the MPI execution—not the actual value provided. Thus, the value associated with "maxprocs" is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. (*End of advice to users.*)

Advice to implementors. High-quality implementations will provide a (key, value) pair for each parameter that can be passed to the command that starts an MPI program. (End of advice to implementors.)

The following function may be used to initialize MPI, and to initialize the MPI thread environment, instead of MPI\_INIT.

MPI\_INIT\_THREAD(required, provided)INrequiredOUTprovidedDUTprovidedDUTprovided

### C binding

int MPI\_Init\_thread(int \*argc, char \*\*\*argv, int required, int \*provided)

```
Fortran 2008 binding
```

```
MPI_Init_thread(required, provided, ierror)
    INTEGER, INTENT(IN) :: required
    INTEGER, INTENT(OUT) :: provided
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### Fortran binding

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
INTEGER REQUIRED, PROVIDED, IERROR
```

This call initializes MPI in the same way that a call to MPI\_INIT would. In addition, it initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values are listed in increasing order of thread support.

MPI\_THREAD\_SINGLE Only one thread will execute.

- MPI\_THREAD\_FUNNELED The process may be multithreaded, but the application must ensure that only the main thread makes MPI calls (for the definition of main thread, see MPI\_IS\_THREAD\_MAIN on page 493).
- MPI\_THREAD\_SERIALIZED The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").

```
MPI_THREAD_MULTIPLE Multiple threads may call MPI, with no restrictions.
```

 $45 \\ 46$ 

<sup>1</sup> These values are monotonic; i.e.,  $MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED <$ 

 $^{2}$  MPI\_THREAD\_SERIALIZED < MPI\_THREAD\_MULTIPLE.

<sup>3</sup> Different processes in MPI\_COMM\_WORLD may require different levels of thread sup <sup>4</sup> port.

The call returns in provided information about the actual level of thread support that
 will be provided by MPI. It can be one of the four values listed above.

The level(s) of thread support that can be provided by MPI\_INIT\_THREAD will depend on the implementation, and may depend on information provided by the user before the program started to execute (e.g., with arguments to mpiexec). If possible, the call will return provided = required. Failing this, the call will return the least supported level such that provided > required (thus providing a stronger level of support than required by the user). Finally, if the user requirement cannot be satisfied, then the call will return in provided the highest supported level.

<sup>14</sup> A thread compliant MPI implementation will be able to return provided <sup>15</sup> = MPI\_THREAD\_MULTIPLE. Such an implementation may always return provided <sup>16</sup> = MPI\_THREAD\_MULTIPLE, irrespective of the value of required.

<sup>17</sup> An MPI library that is not thread compliant must always return provided =

<sup>18</sup> MPI\_THREAD\_SINGLE, even if MPI\_INIT\_THREAD is called on a multithreaded process.
 <sup>19</sup> The library should also return correct values for the MPI calls that can be executed before
 <sup>20</sup> initialization, even if multiple threads have been spawned.

- *Rationale.* Such code is erroneous, but if the MPI initialization is performed by a library, the error cannot be detected until MPI\_INIT\_THREAD is called. The requirements in the previous paragraph ensure that the error can be properly detected. (*End of rationale.*)
- 25 26

21

22

23 24

27 28 A call to MPI\_INIT has the same effect as a call to MPI\_INIT\_THREAD with a required = MPI\_THREAD\_SINGLE.

Vendors may provide (implementation dependent) means to specify the level(s) of 29thread support available when the MPI program is started, e.g., with arguments to mpiexec. 30 This will affect the outcome of calls to MPI\_INIT and MPI\_INIT\_THREAD. Suppose, for  $^{31}$ example, that an MPI program has been started so that only MPI\_THREAD\_MULTIPLE is 32 available. Then MPI\_INIT\_THREAD will return provided = MPI\_THREAD\_MULTIPLE, irre-33 spective of the value of required; a call to MPI\_INIT will also initialize the MPI thread support 34level to MPI\_THREAD\_MULTIPLE. Suppose, instead, that an MPI program has been started 35 so that all four levels of thread support are available. Then, a call to MPI\_INIT\_THREAD 36 will return provided = required; alternatively, a call to MPI\_INIT will initialize the MPI 37 thread support level to MPI\_THREAD\_SINGLE. 38

39

Various optimizations are possible when MPI code is executed single-Rationale. 40threaded, or is executed on multiple threads, but not concurrently: mutual exclusion 41 code may be omitted. Furthermore, if only one thread executes, then the MPI library 42can use library functions that are not thread safe, without risking conflicts with user 43 threads. Also, the model of one communication thread, multiple computation threads 44fits many applications well, e.g., if the process code is a sequential Fortran/C program 45with MPI calls that has been parallelized by a compiler for execution on an SMP node, 46in a cluster of SMPs, then the process computation is multithreaded, but MPI calls 47 will likely execute on a single thread. 48

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multithreaded MPI codes. (End of rationale.)

Advice to implementors. If provided is not MPI\_THREAD\_SINGLE then the MPI library should not invoke C or Fortran library calls that are not thread safe, e.g., in an environment where malloc is not thread safe, then malloc should not be used by the MPI library.

Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI\_INIT\_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time.

Note that required need not be the same value on all processes of MPI\_COMM\_WORLD. (End of advice to implementors.)

As with MPI\_INIT, discussed in Section 11.2.1, the version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL for both arguments.

The following function can be used to query the current level of thread support.

MPI_QUI	ERY_THREAD(provide	ed)			
OUT	provided	provided level of thread support (integer)			
C binding int MPI_Query_thread(int *provided)					
Fortran 2008 binding					
MPI_Query_thread(provided, ierror)					
INTE	EGER, INTENT(OUT)	:: provided			
INTE	EGER, OPTIONAL, IN	TENT(OUT) :: ierror			

### Fortran binding

MPI\_QUERY\_THREAD(PROVIDED, IERROR) INTEGER PROVIDED, IERROR

The call returns in provided the current level of thread support, which will be the value returned in provided by MPI\_INIT\_THREAD, if MPI was initialized by a call to MPI\_INIT\_THREAD(). This function is only applicable when using the World Model to initialize MPI. In the case of applications using both the World Model and the Sessions Model, this function only returns the thread support level returned in provided by MPI\_INIT\_THREAD.

 $\overline{7}$ 

```
1
      MPI_IS_THREAD_MAIN(flag)
2
        OUT
                  flag
                                               true if calling thread is main thread, false otherwise
3
                                               (logical)
4
5
      C binding
6
     int MPI_Is_thread_main(int *flag)
7
8
     Fortran 2008 binding
9
     MPI_Is_thread_main(flag, ierror)
10
          LOGICAL, INTENT(OUT) :: flag
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
      Fortran binding
13
     MPI_IS_THREAD_MAIN(FLAG, IERROR)
14
          LOGICAL FLAG
15
          INTEGER IERROR
16
17
          This function can be called by a thread to determine if it is the main thread (the thread
18
      that called MPI_INIT or MPI_INIT_THREAD). This function is only applicable when using
19
      the World Model to initialize MPI. In the case of applications using both the World Model
20
      and the Sessions Model, the behavior of this procedure is the same as if the application
21
      were only using the World Model.
22
          All routines listed in this section must be supported by all MPI implementations.
23
^{24}
                         MPI libraries are required to provide these calls even if they do not
           Rationale.
25
           support threads, so that portable code that contains invocations to these functions
26
           can link correctly. MPI_INIT continues to be supported so as to provide compatibility
27
           with current MPI codes. (End of rationale.)
28
                                It is possible to spawn threads before MPI is initialized, but
29
           Advice to users.
30
           MPI_COMM_WORLD and MPI_COMM_SELF cannot be used until the World Model is
           active, i.e., until MPI_INIT_THREAD is invoked by one thread (which, thereby, be-
31
           comes the main thread). In particular, it is possible to enter the MPI execution with
32
33
           a multithreaded process.
34
           In the World Model, the level of thread support provided is a global property of the
35
           MPI process that can be specified only once, when MPI is initialized on that process (or
36
           before). Portable third party libraries have to be written so as to accommodate any
37
           provided level of thread support. Otherwise, their usage will be restricted to specific
38
           level(s) of thread support. If such a library can run only with specific level(s) of thread
39
           support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be
40
           used to check whether the user initialized MPI to the correct level of thread support.
41
           (End of advice to users.)
42
43
      11.2.2 Finalizing MPI
44
45
46
      MPI_FINALIZE()
47
48
```

C binding int MPI_Finalize(void)
Fortran 2008 binding MPI_Finalize(ierror) INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding

MPI\_FINALIZE(IERROR) INTEGER IERROR

This routine cleans up all MPI state associated with the World Model. If an MPI program terminates normally (i.e., not due to a call to MPI\_ABORT or an unrecoverable error) then each process must call MPI\_FINALIZE before it exits.

Before an MPI process invokes MPI\_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications associated with the World Model. It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a nonblocking send, it must eventually call MPI\_WAIT, MPI\_TEST, MPI\_REQUEST\_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to MPI\_FINALIZE does not clean up MPI state associated with objects created using MPI\_SESSION\_INIT and other Sessions Model methods, nor objects created using the communicator returned by MPI\_COMM\_GET\_PARENT. See Sections 11.3 and 11.8.

The call to MPI\_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI\_XXX\_FREE calls.

MPI\_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI\_COMM\_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 11.10.4.

The following examples illustrate these rules.

Exan	<b>Example 11.2</b> The following code is correct			32
	•			33
	Process O	Process 1		34
				35
	<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>		36
	<pre>MPI_Send(dest=1);</pre>	<pre>MPI_Recv(src=0);</pre>		37
	<pre>MPI_Finalize();</pre>	<pre>MPI_Finalize();</pre>		38
				39
T				40

**Example 11.3** Without a matching receive, the program is erroneous

Process 0 \_\_\_\_\_ MPI\_Init(); MPI\_Send (dest=1); MPI\_Finalize();

Process 1 \_\_\_\_\_ MPI\_Init(); MPI\_Finalize();

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31 32 33

41 42

43

44

45

46

**Example 11.4** This program is correct: Process 0 calls MPI\_Finalize after it has executed the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call that completes the matching receive operation before it calls MPI\_Finalize.

Process 0	Process 1
<pre>MPI_Init();</pre>	<pre>MPI_Init();</pre>
<pre>MPI_Isend(dest=1);</pre>	<pre>MPI_Recv(src=0);</pre>
<pre>MPI_Request_free();</pre>	<pre>MPI_Finalize();</pre>
<pre>MPI_Finalize();</pre>	<pre>exit();</pre>
<pre>exit();</pre>	

 $\mathbf{2}$ 

**Example 11.5** This program is correct. The attached buffer is a resource allocated by the user, not by MPI; it is available to the user after MPI is finalized.

**Example 11.6** This program is correct. The cancel operation must succeed, since the send cannot complete normally. The wait operation, after the call to MPI\_Cancel, is local—no matching MPI call is required on process 1. Cancelling a send request by calling MPI\_CANCEL is deprecated.

```
Process 0 Process 1
-----
MPI_Issend(dest=1); MPI_Finalize();
MPI_Cancel();
MPI_Wait();
MPI_Finalize();
```

Advice to implementors. Even though a process has executed all MPI calls needed to complete the communications it is involved with, such communication may not yet be completed from the viewpoint of the underlying MPI system. For example, a blocking send may have returned, even though the data is still buffered at the sender in an MPI buffer; an MPI process may receive a cancel request for a message it has completed receiving. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI\_FINALIZE returns. Thus, if a process exits after the call to MPI\_FINALIZE, this will not cause an ongoing communication to fail. The MPI implementation should also complete freeing all objects marked for deletion by MPI calls that freed them. (*End of advice to implementors.*)

Failures may disrupt MPI operations during and after MPI finalization. A high quality implementation shall not deadlock in MPI finalization, even in the presence of failures. The normal rules for MPI error handling continue to apply. After MPI\_COMM\_SELF has been "freed" (see Section 11.2.4), errors that are not associated with a communicator, window, or file raise the initial error handler (set during the launch operation, see 11.8.4).

Although it is not required that all processes return from MPI\_FINALIZE, it is required that, when it has not failed or aborted, at least the MPI process that was assigned rank 0 in MPI\_COMM\_WORLD returns, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, users may desire to supply an exit code for each process that returns from MPI\_FINALIZE.

Note that a failure may terminate the MPI process that was assigned rank 0 in MPI\_COMM\_WORLD, in which case it is possible that no MPI process returns from MPI\_FINALIZE.

Advice to users. Applications that handle errors are encouraged to implement all rank-specific code before the call to MPI\_FINALIZE. In Example 11.7 below, the process with rank 0 in MPI\_COMM\_WORLD may have been terminated before, during, or after the call to MPI\_FINALIZE, possibly leading to the code after MPI\_FINALIZE never being executed. (*End of advice to users.*)

**Example 11.7** The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile", "w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);
```

11.2.3 Determining Whether MPI Has Been Initialized When Using the World Model

One of the goals of MPI is to allow for layered libraries. A library using the World Model needs to know if MPI has been initialized using either of MPI\_INIT or MPI\_INIT\_THREAD. In MPI the function MPI\_INITIALIZED is provided to tell if MPI had been initialized using the World Model. In the World Model, once MPI has been finalized it cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this, the function MPI\_FINALIZED is needed.

 $\mathbf{2}$ 

```
1
     MPI_INITIALIZED(flag)
2
        OUT
                 flag
                                              Flag is true if MPI_INIT has been called and false
3
                                              otherwise (logical)
4
5
      C binding
6
     int MPI_Initialized(int *flag)
\overline{7}
8
      Fortran 2008 binding
9
     MPI_Initialized(flag, ierror)
10
          LOGICAL, INTENT(OUT) :: flag
11
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
      Fortran binding
13
     MPI_INITIALIZED(FLAG, IERROR)
14
          LOGICAL FLAG
15
          INTEGER IERROR
16
17
          This routine may be used to determine whether MPI_INIT or MPI_INIT_THREAD has
18
      been called. MPI_INITIALIZED returns true if the calling process has called either of these
19
      MPI procedures. Whether MPI_FINALIZE has been called does not affect the behavior of
20
      MPI_INITIALIZED. This function must always be thread-safe, as defined in Section 11.6.
21
      This function returns false for applications using the Sessions Model exclusively.
22
23
      MPI_FINALIZED(flag)
^{24}
25
       OUT
                                              true if MPI was finalized (logical)
                 flag
26
27
      C binding
28
      int MPI_Finalized(int *flag)
29
30
     Fortran 2008 binding
     MPI_Finalized(flag, ierror)
^{31}
32
          LOGICAL, INTENT(OUT) :: flag
33
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
      Fortran binding
35
     MPI_FINALIZED(FLAG, IERROR)
36
          LOGICAL FLAG
37
          INTEGER IERROR
38
39
          This routine returns true if MPI_FINALIZE has completed. It is valid to call
40
      MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE. This function must always be
41
      thread-safe, as defined in Section 11.6.
42
43
      11.2.4 Allowing User Functions at MPI Finalization
44
     In the context of the World Model, there are times in which it would be convenient to
45
     have actions happen when an MPI process finalizes MPI. For example, a routine may do
46
      initializations that are useful until the MPI job (or that part of the job that is being termi-
47
      nated in the case of dynamically created processes) finalizes MPI. This can be accomplished
```

in MPI by attaching an attribute to MPI\_COMM\_SELF with a callback function. When MPI\_FINALIZE is called, it will first execute the equivalent of an MPI\_COMM\_FREE on MPI\_COMM\_SELF. This will cause the delete callback function to be executed on all keys associated with MPI\_COMM\_SELF, in the reverse order that they were set on MPI\_COMM\_SELF. If no key has been attached to MPI\_COMM\_SELF, then no callback is invoked. The "freeing" of MPI\_COMM\_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI\_FINALIZED will return false in any of these callback functions. Once done with MPI\_COMM\_SELF, the order and rest of the actions taken by MPI\_FINALIZE is not specified.

Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI\_COMM\_SELF internally should register their internal callbacks before returning from MPI\_INIT / MPI\_INIT\_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

### 11.3 The Sessions Model

There are a number of limitations with the World Model described in the preceding section. Among these are the following: MPI cannot be initialized from different application components without *a priori* knowledge or coordination; MPI cannot be initialized more than once; and MPI cannot be reinitialized after MPI\_FINALIZE has been called. This section describes an alternative approach to MPI initialization—the Sessions Model. With this approach, an MPI application, or components of the application, can instantiate MPI resources for the specific communication needs of this component. MPI\_COMM\_WORLD is not valid for use as a communicator. MPI\_INFO\_ENV is not valid for use as an info object when only using the Sessions Model. As described in Section 11.2.1, MPI must be initialized using the World Model to use this info object. Note that an application may employ both the Sessions Model and World Model concurrently (see Section 11.1).

In the Sessions Model, MPI resources can be allocated and freed multiple times in an MPI process.

33 As shown in Figure 11.1, when using the Sessions Model, an MPI process instantiates 34an MPI Session handle, which can be used to query the runtime system about character-35 istics of the job within which the process is running, as well as other system resources. Using this information, the MPI process can then create an MPI Group based on appli-36 37 cation requirements and available resources, which in turn can be used to create an MPI Communicator, Window, or File. By judicious creation of communicators, an application 3839 only needs to allocate MPI resources based on its communication requirements. Although there are existing MPI interfaces for creating communicators which can, in principle, allow 40 41 for resource optimizations within an MPI implementation, this can only be done following 42initialization of MPI.

For multithreaded applications, the Sessions Model provides fine-grain control of the thread support level for MPI objects. It is possible to specify different thread support levels when creating different *MPI Session handles*. Thus different components of an application can use different thread support levels.

The Sessions Model introduces a concept of isolation. MPI objects derived from different *MPI Session handles* shall not be intermixed with each other in a single MPI procedure 48

1 2

3

4

 $\mathbf{5}$ 

6 7

8 9

10

11

12

13

14

15

16 17 18

19 20

21

22

23 24

25

26

27 28

29

30

31

32

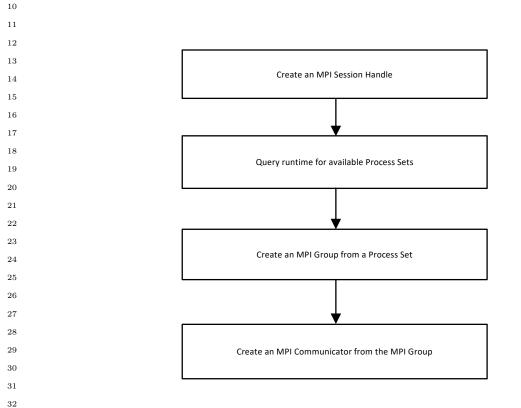
43

44

 $45 \\ 46$ 

call. MPI objects derived from the Sessions Model shall not be intermixed in a single MPI
 procedure call with MPI objects derived from the World Model. MPI objects derived from
 the Sessions Model shall not be intermixed in a single MPI procedure call with MPI objects derived from the communicator obtained from a call to MPI\_COMM\_GET\_PARENT
 or MPI\_COMM\_JOIN.

This restriction does not apply to generalized requests (Section 13.2) as such requests are not associated directly with communicators or other MPI objects. Note however, the Sessions Model does not otherwise change the semantics or behavior of MPI objects.



6

7

8

9

33 34

Figure 11.1: Steps to creating an MPI Communicator from an MPI Session handle.

11.3.1	Session Creation and De	estruction Methods	1		
-			2		
			3		
MPI SES	SION_INIT(info, errhand	ler. session)	4		
	Ϋ́Υ.	,	5		
IN	info	info object to specify thread support level and MPI	6		
		implementation specific resources (handle)	7		
IN	errhandler	error handler to invoke in the event that an error is	8		
		encountered during this function call (handle)	9		
OUT	session	new session (handle)	10		
			11		
Chindi			12		
C bindi	0	a infa MDT Eachandler annhandler	13		
int MPI_		to info, MPI_Errhandler errhandler,	14		
	MPI_Session *s	ession)	15		
Fortran	2008 binding		16		
MPI_Sess	sion_init(info, errha	andler, session, ierror)	17		
TYPE	E(MPI_Info), INTENT()	IN) :: info	18		
TYPE	TYPE(MPI_Errhandler), INTENT(IN) :: errhandler				
TYPE	TYPE(MPI_Session), INTENT(OUT) :: session				
INTE	EGER, OPTIONAL, INTEN	NT(OUT) :: ierror	21		
Dentes	1 . 1.		22		
Fortran binding					

MPI_SESSION_	INIT(	INFO,	ERRHAN	DLER,	SES	SION,	IERROR)
INTEGER	INFO,	ERRH	ANDLER,	SESSI	CON,	IERRO	DR

The info argument is used to request MPI functionality requirements and possible MPI implementation specific capabilities. The following info key is predefined:

"thread\_level" used to request the thread support level required for MPI objects derived from the Session. Allowed values are "MPI\_THREAD\_SINGLE", "MPI\_THREAD\_FUNNELED", "MPI\_THREAD\_SERIALIZED", and "MPI\_THREAD\_MULTIPLE". Note that the thread support value is specified by a string rather than the integer values supplied to MPI\_INIT\_THREAD. The thread support level actually provided by the MPI implementation can be determined via a subsequent call to MPI\_SESSION\_GET\_INFO to return the info object associated with the Session. The default thread support level is MPI implementation dependent.

The errhandler argument specifies an error handler to invoke in the event that the Session instantiation call encounters an error. The error handler shall be either a pre-defined error handler (see 9.3) or one created using MPI\_SESSION\_CREATE\_ERRHANDLER. Session instantiation is intended to be a lightweight operation. An MPI process may instantiate multiple Sessions. MPI\_SESSION\_INIT is always thread safe; multiple threads within an application may invoke it concurrently.

Advice to users. Requesting "MPI\_THREAD\_SINGLE" thread support level is generally not recommended, because this will conflict with other components of an application requesting higher levels of thread support. (End of advice to users.)

 $^{24}$ 

 $^{31}$ 

1 Advice to implementors. Owing to the restrictions of the MPI\_THREAD\_SINGLE  $\mathbf{2}$ thread support level, implementators are discouraged from making this the default 3 thread support level for Sessions. (End of advice to implementors.) 4 56 MPI\_SESSION\_FINALIZE(session) 7 8 INOUT session session to be finalized (handle) 9 10 C binding 11int MPI\_Session\_finalize(MPI\_Session \*session) 12Fortran 2008 binding 13 MPI\_Session\_finalize(session, ierror) 14TYPE(MPI\_Session), INTENT(INOUT) :: session 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 Fortran binding 18 MPI\_SESSION\_FINALIZE(SESSION, IERROR) 19INTEGER SESSION, IERROR 20This routine cleans up all MPI state associated with the supplied session. Every instantiated 21Session must be finalized using MPI\_SESSION\_FINALIZE. The handle session is set to 22 MPI\_SESSION\_NULL by the call. 23Before an MPI process invokes MPI\_SESSION\_FINALIZE, the process must perform  $^{24}$ all MPI calls needed to complete its involvement in MPI communications: it must locally 25complete all MPI operations that it initiated and it must execute matching calls needed to 26complete MPI communications initiated by other processes. 27The call to MPI\_SESSION\_FINALIZE does not free objects created by MPI calls; these 28objects are freed using MPI\_XXX\_FREE calls. 29MPI\_SESSION\_FINALIZE may be synchronizing on any or all of the groups associated 30 with communicators, windows, or files derived from the session and not disconnected, freed,  $^{31}$ or closed, respectively, before the call to the MPI\_SESSION\_FINALIZE procedure. 32 MPI\_SESSION\_FINALIZE behaves as if all such synchronizations occur concurrently. As 33 MPI\_COMM\_FREE may mark a communicator for freeing later, MPI\_SESSION\_FINALIZE 34may be synchronizing on the group associated with a communicator that is only freed (with 35 MPI\_COMM\_FREE) rather than disconnected (with MPI\_COMM\_DISCONNECT). 36 37 *Rationale.* This rule is similar to the rule that MPI\_FINALIZE is collective (see 11.2.2), 38 but does not require that MPI\_SESSION\_FINALIZE be collective over all connected 39 MPI processes. It also allows for cases where some MPI processes may have derived a 40 set of communicators using a different number of session handles. See Example 11.8. 41 (End of rationale.) 4243 Advice to implementors. This rule also allows for the completion of communications 44the MPI process is involved with that may not yet be completed from the viewpoint 45of the underlying MPI system. See the advice to implementors at the end of Sec-46tion 11.2.2. (End of advice to implementors.) 47

Advice to implementors. An MPI implementation should be able to implement the semantics of MPI\_SESSION\_FINALIZE as a *local* procedure, provided an application frees all MPI windows, closes all MPI files, and uses MPI\_COMM\_DISCONNECT to free all MPI communicators associated with a session prior to invoking MPI\_SESSION\_FINALIZE on the corresponding session handle. (*End of advice to implementors.*)

**Example 11.8** Three MPI processes are connected with 2 communicators (indicated by the = symbols), derived from one session handle in process X but from two separate session handles in both process Y and Z.

process-X	process-Y	process-Z	Remarks
			sesX, sesYA, ses YB, sesZA and
			sesZB are session handles.
(sesX)====	====(sesYA)====	===(sesZA)	communicator_1 and
(sesX)====	====(sesYB)====	===(sesZB)	communicator_2 are derived
			from them.
SF(sesX)	SF(sesYA)	SF(sesZA)	SF = MPI_SESSION_FINALIZE
	SF(sesYB)	SF(sesZB)	

Process X has only to finalize its one session handle, whereas the other two MPI processes have to call MPI\_SESSION\_FINALIZE twice in the same sequence with respect to the communicators derived from the session handles. Specifically, both process Y and process Z shall call MPI\_SESSION\_FINALIZE for the session from which communicator\_1 was derived before calling the MPI\_SESSION\_FINALIZE for the session from which communicator\_2 was derived, or vice versa (i.e. both shall finalize the session for communicator\_2 first then finalize the session for communicator\_1). The call SF(ses) in process X may not return until both SF(ses\*A) and SF(ses\*B) are called in processes Y and Z.

# 11.3.2 Processes Sets

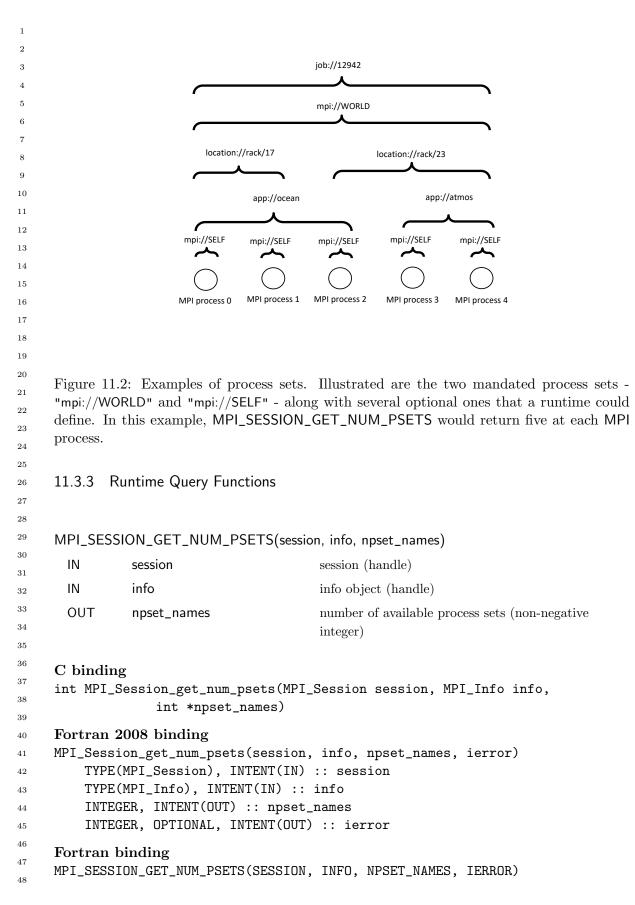
Process sets are the mechanism for MPI applications to query the runtime. Process sets are identified by process set names. Process set names have a *Uniform Resource Identifier* (URI) format. Two process set names are mandated: "mpi://WORLD" and "mpi://SELF". Additional process set names may be defined, for example, "mpix://UNIVERSE" and "hwloc://L3Cache" may be defined by the MPI implementation. The "mpi://" namespace is reserved for exclusive use by the MPI standard. Figure 11.2 depicts process sets that the runtime could associate with an instance of an MPI job. In this example, the two mandated process sets are defined, in addition to optional, implementation specific ones.

Mechanisms for defining process sets and how system resources are assigned to these sets is considered to be implementation dependent.

A process set caches key/value tuples that are accessible to the application via an MPI\_Info object. The "mpi\_size" key is mandatory for all process sets.

 $\mathbf{5}$ 

 $^{24}$ 



#### INTEGER SESSION, INFO, NPSET\_NAMES, IERROR

This function is used to query the runtime for the number of available process sets in which the calling MPI process is a member. An MPI implementation is allowed to increase the number of available process sets during the execution of an MPI application when new process sets become available. However, MPI implementations are not allowed to change the index of a particular process set name, or to change the name of the process set at a particular index, or to delete a process set name once it has been added. When a process set becomes invalid, for example, when some processes become unreachable due to failures in the communication system, subsequent usage of the process set name should raise an error. For example, creating an MPI\_Group from such a process set might succeed because it is a local operation, but creating an MPI\_Comm from that group and attempting collective communication should raise an error.

Advice to implementors. It is anticipated that an MPI implementation may be relying on an external runtime system to provide process sets. Such runtime systems may have the ability to dynamically create process sets during the course of application execution. Requiring the number of process sets returned by MPI\_SESSION\_GET\_NUM\_PSETS to be constant over the course of application execution would prevent an application from taking advantage of such capabilities. (*End* of advice to implementors.)

MPI_SESSION_	GET_NTH_PSET(session,	n, info, n, pset_len, pset_name)	
	(	, , , , , , , , , , , , , , , , , , , ,	

IN	session	session (handle)
IN	info	info object (handle)
IN	n	index of the desired process set name (integer)
INOUT	pset_len	length of the pset_name argument (integer)
OUT	pset_name	name of the nth process set (string)

### C binding

### Fortran 2008 binding

MPI\_Session\_get\_nth\_pset(session, info, n, pset\_len, pset\_name, ierror)
 TYPE(MPI\_Session), INTENT(IN) :: session
 TYPE(MPI\_Info), INTENT(IN) :: info
 INTEGER, INTENT(IN) :: n
 INTEGER, INTENT(INOUT) :: pset\_len
 CHARACTER(LEN=\*), INTENT(OUT) :: pset\_name
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

### Fortran binding

MPI\_SESSION\_GET\_NTH\_PSET(SESSION, INFO, N, PSET\_LEN, PSET\_NAME, IERROR)
INTEGER SESSION, INFO, N, PSET\_LEN, IERROR
CHARACTER\*(\*) PSET\_NAME

 $\mathbf{2}$ 

# 506 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
          This function returns the name of the nth process set in the supplied pset_name buffer.
\mathbf{2}
      pset_len is the size of the buffer needed to store the nth process set name. If the pset_len
3
      passed into the function is less than the actual buffer size needed for the process set name,
4
      then the string value returned in pset_name is truncated. If pset_len is set to 0, pset_name is
\mathbf{5}
      not changed. On return, the value of pset_len will be set to the required buffer size to hold
6
      the process set name. In C, pset_len includes the required space for the null terminator. In
\overline{7}
      C, this function returns a null terminated string in all cases where the pset_len input value
8
     is greater than 0.
9
          If two MPI processes get the same process set name, then the intersection of the two
10
      process sets shall either be the empty set or identical to the union of the two process sets.
11
          After a successful call to MPI_SESSION_GET_NTH_PSET, subsequent calls to routines
12
      that query information about the same process set name and same session handle must
13
      return the same information. An MPI implementation is not allowed to alter any of the
14
      returned process set names.
15
          Process set names have an implementation-defined maximum length of
16
      MPI_MAX_PSET_NAME_LEN characters. MPI_MAX_PSET_NAME_LEN shall have a value of
17
      at least 63.
18
                              MPI_MAX_PSET_NAME_LEN might be very large, so it might not
19
           Advice to users.
           be wise to declare a string of that size. Users are encouraged to use
20
           MPI_SESSION_GET_NTH_PSET both for obtaining the length of a pset_name and
21
           the process set name. (End of advice to users.)
22
23
^{24}
25
      MPI_SESSION_GET_INFO(session, info_used)
26
       IN
                 session
                                               session (handle)
27
28
       OUT
                 info_used
                                               see explanation below (handle)
29
30
      C binding
^{31}
      int MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)
32
      Fortran 2008 binding
33
34
     MPI_Session_get_info(session, info_used, ierror)
          TYPE(MPI_Session), INTENT(IN) :: session
35
          TYPE(MPI_Info), INTENT(OUT) :: info_used
36
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     Fortran binding
39
     MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR)
40
          INTEGER SESSION, INFO_USED, IERROR
41
          MPI_SESSION_GET_INFO returns a new info object containing the hints of the MPI
42
      Session associated with session. The current setting of all hints related to this MPI Session
43
      is returned in info_used. An MPI implementation is required to return all hints that are
44
      supported by the implementation and have default values specified; any user-supplied hints
45
46
      that were not ignored by the implementation; and any additional hints that were set by
47
      the implementation. If no such hints exist, a handle to a newly created info object is
```

returned that contains no key/value pair. The user is responsible for freeing info\_used via MPI\_INFO\_FREE.

MPI_SES	MPI_SESSION_GET_PSET_INFO(session, pset_name, info)					
IN	session	session (handle)				
IN	pset_name	name of process set (string)				
OUT	info	info object containing information about the given process set (handle)				
	C binding int MPI_Session_get_pset_info(MPI_Session session, const char *pset_name, MPI_Info *info)					
MPI_Ses TYP CHA TYP	<pre>Fortran 2008 binding MPI_Session_get_pset_info(session, pset_name, info, ierror)     TYPE(MPI_Session), INTENT(IN) :: session     CHARACTER(LEN=*), INTENT(IN) :: pset_name     TYPE(MPI_Info), INTENT(OUT) :: info     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>					
Fortran binding       2         MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR)       2         INTEGER SESSION, INFO, IERROR       2         CHARACTER*(*) PSET_NAME       2						
This function is used to query properties of a specific process set. The returned <i>info</i> object can be queried with existing MPI info object query functions. One key/value pair must be defined, "mpi_size". The value of the "mpi_size" key specifies the number of MPI						

# 11.3.4 Sessions Model Examples

This section presents several examples of how to use MPI Sessions to create MPI Groups and MPI Communicators.

processes in the process set. The user is responsible for freeing the returned MPI\_Info object.

**Example 11.9** Simple example illustrating creation of an MPI communicator using the Sessions Model.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
static MPI_Session lib_shandle = MPI_SESSION_NULL;
static MPI_Comm lib_comm = MPI_COMM_NULL;
int library_foo_init(void)
{
```

 $^{41}$ 

 $45 \\ 46$ 

```
1
        int rc, flag, valuelen;
\mathbf{2}
        int ret = 0;
3
        const char pset_name[] = "mpi://WORLD";
4
        const char mt_key[] = "thread_level";
5
        const char mt_value[] = "MPI_THREAD_MULTIPLE";
6
        char out_value[100]; /* large enough */
7
        MPI_Group wgroup = MPI_GROUP_NULL;
8
        MPI_Info sinfo = MPI_INFO_NULL;
9
        MPI_Info tinfo = MPI_INFO_NULL;
10
^{11}
        MPI_Info_create(&sinfo);
12
        MPI_Info_set(sinfo, mt_key, mt_value);
13
        rc = MPI_Session_init(sinfo, MPI_ERRORS_RETURN,
14
                                 &lib_shandle);
15
        if (rc != MPI_SUCCESS) {
16
            ret = -1;
17
            goto fn_exit;
18
        }
19
20
        /*
21
         * check we got thread support level foo library needs
22
         */
23
        rc = MPI_Session_get_info(lib_shandle, &tinfo);
^{24}
        if (rc != MPI_SUCCESS) {
25
            ret = -1;
26
            goto fn_exit;
27
        }
28
29
        valuelen = sizeof(out_value);
30
        MPI_Info_get_string(tinfo, mt_key, &valuelen,
31
                       out_value, &flag);
32
        if (0 == flag) {
33
            printf("Could not find key %s\n", mt_key);
34
            ret = -1;
35
            goto fn_exit;
36
        }
37
38
        if (strcmp(out_value, mt_value)) {
39
            printf("Did not get thread multiple support, got %s\n",
40
                   out_value);
41
            ret = -1;
42
            goto fn_exit;
43
        }
44
45
         /*
46
         * create a group from the WORLD process set
47
         */
48
```

```
1
   rc = MPI_Group_from_session_pset(lib_shandle,
                                                                                         \mathbf{2}
                                        pset_name,
                                                                                         3
                                        &wgroup);
                                                                                         4
   if (rc != MPI_SUCCESS) {
                                                                                         5
      ret = -1;
                                                                                         6
      goto fn_exit;
                                                                                         7
   }
                                                                                         8
                                                                                         9
   /*
                                                                                         10
    * get a communicator
                                                                                         11
    */
                                                                                         12
   rc = MPI_Comm_create_from_group(wgroup,
                                                                                         13
                                       "org.mpi-forum.mpi-v4_0.example-ex11_8",
                                                                                         14
                                       MPI_INFO_NULL,
                                                                                         15
                                       MPI_ERRORS_RETURN,
                                                                                         16
                                       &lib_comm);
                                                                                         17
   if (rc != MPI_SUCCESS) {
                                                                                         18
      ret = -1;
                                                                                         19
      goto fn_exit;
                                                                                         20
   }
                                                                                         21
                                                                                         22
   /*
                                                                                         23
    * free group, library doesn't need it.
                                                                                         24
    */
                                                                                         25
                                                                                         26
fn_exit:
                                                                                         27
   MPI_Group_free(&wgroup);
                                                                                         28
                                                                                         29
   if (sinfo != MPI_INFO_NULL) {
                                                                                         30
      MPI_Info_free(&sinfo);
                                                                                         31
   }
                                                                                         32
                                                                                         33
   if (tinfo != MPI_INFO_NULL) {
                                                                                         34
      MPI_Info_free(&tinfo);
                                                                                         35
   }
                                                                                         36
                                                                                         37
   if (ret != 0) {
                                                                                         38
      MPI_Session_finalize(&lib_shandle);
                                                                                         39
   }
                                                                                         40
                                                                                         41
   return ret;
                                                                                         42
}
                                                                                         43
```

Example 11.9 shows how the pre-defined "mpi://WORLD" process set can be used to first create a local MPI group and then subsequently to create an MPI communicator from this group.

44

45

```
1
     Example 11.10 This example illustrates the use of Process Set query functions to select
\mathbf{2}
     a Process Set to use for MPI Group creation.
3
4
     #include <stdio.h>
     #include <stdlib.h>
5
6
     #include <string.h>
7
     #include "mpi.h"
8
9
     int main(int argc, char *argv[])
10
     {
         int i, n_psets, psetlen, rc, ret;
11
         int valuelen;
12
         int flag = 0;
13
14
         char *pset_name = NULL;
         char *info_val = NULL;
15
16
        MPI_Session shandle = MPI_SESSION_NULL;
17
        MPI_Info sinfo = MPI_INFO_NULL;
        MPI_Group pgroup = MPI_GROUP_NULL;
18
19
         if (argc < 2) {
20
            fprintf(stderr, "A process set name fragment is required\n");
21
            return EXIT_FAILURE;
22
         }
23
^{24}
         rc = MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &shandle);
25
26
         if (rc != MPI_SUCCESS) {
            fprintf(stderr, "Could not initialize session, bailing out\n");
27
            return EXIT_FAILURE;
28
         }
29
30
         MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, &n_psets);
31
32
         for (i=0, pset_name=NULL; i<n_psets; i++) {</pre>
33
34
             psetlen = 0;
             MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
35
                                        &psetlen, NULL);
36
37
             pset_name = (char *)malloc(sizeof(char) * psetlen);
             MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, i,
38
                                        &psetlen, pset_name);
39
             if (strstr(pset_name, argv[1]) != NULL) break;
40
41
42
             free(pset_name);
             pset_name = NULL;
43
         }
44
45
         /*
46
47
          * get instance of an info object for this Session
48
```

```
*/
   MPI_Session_get_pset_info(shandle, pset_name, &sinfo);
   valuelen = 0;
   MPI_Info_get_string(sinfo, "mpi_size", &valuelen, NULL, &flag);
   if (flag) {
       info_val = (char *)malloc(valuelen);
       MPI_Info_get_string(sinfo, "mpi_size", &valuelen, info_val, &flag);
       free(info_val);
    }
   /*
    * create a group from the process set
    */
   rc = MPI_Group_from_session_pset(shandle, pset_name,
                                     &pgroup);
   ret = (rc == MPI_SUCCESS) ? 0 : EXIT_FAILURE;
   free(pset_name);
   MPI_Group_free(&pgroup);
   MPI_Info_free(&sinfo);
   MPI_Session_finalize(&shandle);
   fprintf(stderr, "Test completed ret = %d\n", ret);
   return ret;
}
```

Example 11.10 illustrates several aspects of the Sessions Model. First, the default error handler can be specified when instantiating a Session instance. Second, there must be at least two process sets associated with a Session. Third, the example illustrates use of the Sessions info object and the one required key: "mpi\_size".

**Example 11.11** A Fortran 2008 example illustrating how to obtain information about available process sets, create an MPI Group from a process set, and subsequently create an MPI Communicator.

```
PROGRAM MAIN
USE mpi_f08
IMPLICIT NONE
INTEGER :: pset_len, ierror, n_psets
CHARACTER(LEN=:), ALLOCATABLE :: pset_name
TYPE(MPI_Session) :: shandle
TYPE(MPI_Group) :: pgroup
TYPE(MPI_Group) :: pgroup
TYPE(MPI_Comm) :: pcomm
CALL MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &
shandle, ierror)
```

1

2 3

4

5

6

7

8

9

10

11 12

13

14

15 16

17

18

19 20

21

22

23

24 25

26

27

28

29

30

31

32 33

34

35

36 37

38

39

40

41

42

43

44 45 46

47

#### 512 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

```
1
          IF (ierror .NE. MPI_SUCCESS) THEN
2
             WRITE(*,*) "MPI_Session_init failed"
3
             ERROR STOP
4
         END IF
5
6
         CALL MPI_Session_get_num_psets(shandle, MPI_INFO_NULL, n_psets)
7
         IF (n_psets .LT. 2) THEN
8
             WRITE(*,*) "MPI_Session_get_num_psets didn't return at least 2 psets"
9
             ERROR STOP
10
         END IF
11
12
     !
13
     !
         Just get the second pset's length and name
14
     !
         Note that index values are zero-based, even in Fortran
15
     !
16
17
         pset_len = 0
18
         CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                           &
19
                                          pset_len, pset_name)
20
         ALLOCATE(CHARACTER(LEN=pset_len)::pset_name)
21
         CALL MPI_Session_get_nth_pset(shandle, MPI_INFO_NULL, 1,
                                                                           &
22
                                          pset_len, pset_name)
23
^{24}
     ļ
25
     !
         create a group from the pset
26
     !
27
         CALL MPI_Group_from_session_pset(shandle, pset_name, pgroup)
28
     !
29
     !
         free the buffer used for the pset name
30
     !
^{31}
         DEALLOCATE(pset_name)
32
33
     !
34
     !
         create a MPI communicator from the group
35
     !
36
         CALL MPI_Comm_create_from_group(pgroup, "session_example",
                                                                           &
37
                                                     MPI_INFO_NULL,
                                                                           &
38
                                                     MPI_ERRORS_RETURN,
                                                                           &
39
                                                     pcomm)
40
41
         CALL MPI_Barrier(pcomm, ierror)
42
          IF (ierror .NE. MPI_SUCCESS) THEN
43
              WRITE(*,*) "Barrier call on communicator failed"
44
              ERROR STOP
45
         END IF
46
47
         CALL MPI_Comm_free(pcomm)
48
```

```
CALL MPI_Group_free(pgroup)
CALL MPI_Session_finalize(shandle, ierror)
```

```
END PROGRAM MAIN
```

Note in this example that the call to MPI\_SESSION\_FINALIZE may block in order to ensure that the calling MPI process has completed its involvement in the preceding MPI\_BARRIER operation. If MPI\_COMM\_DISCONNECT had been used instead of MPI\_COMM\_FREE, the example would have blocked in MPI\_COMM\_DISCONNECT rather than MPI\_SESSION\_FINALIZE.

# 11.4 Common Elements of Both Process Models

### 11.4.1 MPI Functionality that is Always Available

Some MPI functions may be invoked at any time, including prior to calling MPI\_INIT or MPI\_SESSION\_INIT, and following MPI finalization, independent of whether the World Model, Sessions Model, or both are used. These functions can be called concurrently by multiple threads within an MPI Process. Table 11.1 lists the applicable MPI functions.

MPI_INITIALIZED
MPI_FINALIZED
MPI_GET_VERSION
MPI_GET_LIBRARY_VERSION
MPI_INFO_CREATE
MPI_INFO_CREATE_ENV
MPI_INFO_SET
MPI_INFO_DELETE
MPI_INFO_GET_STRING
MPI_INFO_GET_NKEYS
MPI_INFO_GET_NTHKEY
MPI_INFO_DUP
MPI_INFO_FREE
MPI_INFO_F2C
MPI_INFO_C2F
MPI_SESSION_CREATE_ERRHANDLER
MPI_SESSION_CALL_ERRHANDLER
MPI_ERRHANDLER_FREE
MPI_ERRHANDLER_F2C
MPI_ERRHANDLER_C2F
MPI_ERROR_STRING
MPI_ERROR_CLASS

Table 11.1: List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization

In addition to the functions listed in Table 11.1, any function with the prefix MPI\_T\_

 $45 \\ 46$ 

### 514 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

(within the constraints for functions with this prefix listed in Section 15.3.4) may also be
 called prior to MPI initialization and after MPI finalization.

3					
4 5	11.4.2	Aborting MPI Processes			
6					
7					
8	MPI_A	BORT(comm, errorcode)			
9	IN	comm	communicator of $MPI$ processes to abort (handle)		
10	IN	errorcode	error code to return to invoking environment		
11			(integer)		
12					
13	C bind	ling			
14 15	int MP	I_Abort(MPI_Comm comm, int	errorcode)		
15	Fortra	n 2008 binding			
17	MPI_Abort(comm, errorcode, ierror)				
18	TYPE(MPI_Comm), INTENT(IN) :: comm				
19	INTEGER, INTENT(IN) :: errorcode				
20	IN	TEGER, OPTIONAL, INTENT(OU	T) :: ierror		
21	Fortra	n binding			
22		ORT(COMM, ERRORCODE, IERRO	R)		
23		TEGER COMM, ERRORCODE, IER			
24		· · · · · · · · · · · · · · · · · · ·			

This routine makes a "best attempt" to abort all MPI processes in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a return errorcode from the main program.

It may not be possible for an MPI implementation to abort only the processes rep-29resented by comm if this is a subset of the processes. In this case, the MPI implemen-30 tation should attempt to abort all the connected processes but should not abort any un- $^{31}$ connected processes. When using the World Model, and if no processes were spawned, 32 accepted, or connected then this has the effect of aborting all the processes associated with 33 34MPI\_COMM\_WORLD. In the case of the Sessions Model, if an MPI process has instantiated multiple sessions, the union of the process sets in these sessions are considered connected 35 processes. Thus invoking MPI\_ABORT on a communicator derived from one of these ses-36 sions will result in all MPI processes in this union being aborted. 37

38

3

Advice to implementors. After aborting a subset of processes, a high quality im-39 plementation should be able to provide error handling for communicators, windows, 40 and files involving both aborted and non-aborted processes. As an example, if the 41 user changes the error handler for MPI\_COMM\_WORLD to MPI\_ERRORS\_RETURN or a 42custom error handler, when a subset of MPI\_COMM\_WORLD is aborted, the remaining 43 processes in MPI\_COMM\_WORLD should be able to continue communicating with each 44other and receive an appropriate error code when attempting communication with 45an aborted process (e.g., an error of class MPI\_ERR\_PROC\_ABORTED). A high quality 46implementation should support equivalent behavior for communicators derived from 47 sessions. (End of advice to implementors.) 48

Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. (*End of advice to users.*)

Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)

# 11.5 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

```
mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard startup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of mpirun MPI specifies mpiexec.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an mpiexec startup command and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called mpiexec, it must be of the form described below.

It is suggested that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start <program> with an initial set of <numprocs> processes, which will be accessible as the process set named "mpi://WORLD" in the Sessions Model and/or used to form the group associated with the built-in communicator, MPI\_COMM\_WORLD in the World Model. Other arguments to mpiexec may be implementation-dependent.

Advice to implementors. Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that mpiexec be able to be viewed as a command-line version of MPI\_COMM\_SPAWN (See Section 11.8.4).

Analogous to MPI\_COMM\_SPAWN, we have

mpiexec -n	<maxpi< th=""><th>rocs&gt;</th></maxpi<>	rocs>
-soft	<	>
-host	<	>

1 2

3

4 5

6

7

8 9 10

11

12 13

14

15 16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32 33

34

35

36

37

38 39

40

41

42

43

1 -arch < > 2 < > -wdir 3 < -path > 4 -file < > 5-initial-errhandler < > 6 . . . 7 <command line> 8 9 for the case where a single command line for the application program and its arguments 10 will suffice. See Section 11.8.4 for the meanings of these arguments. For the case 11 corresponding to MPI\_COMM\_SPAWN\_MULTIPLE there are two possible formats: 12Form A: 13 14mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... } 151617 As with MPI\_COMM\_SPAWN, all the arguments are optional. (Even the -n x argu-18 ment is optional; the default is implementation dependent. It might be 1, it might be 19 taken from an environment variable, or it might be specified at compile time.) The names and meanings of the arguments are taken from the keys in the info argument 2021to MPI\_COMM\_SPAWN. There may be other, implementation-dependent arguments 22 as well. 23Note that Form A, though convenient to type, prevents colons from being program 24arguments. Therefore an alternate, file-based form is allowed: 25Form B: 2627mpiexec -configfile <filename> 2829 30 where the lines of *<*filename> are of the form separated by the colons in Form A. 31Lines beginning with '#' are comments, and lines may be continued by terminating 32 the partial line with ' $\$ '. 33 34 **Example 11.12** Start 16 instances of myprog on the current or default machine: 35 36 mpiexec -n 16 myprog 37 38 **Example 11.13** Start 10 instances of myprog on the machine called ferrari: 39 40 mpiexec -n 10 -host ferrari myprog 41 4243 **Example 11.14** Start 3 instances of the same program myprog with different com-44 mand-line arguments: 45 46 mpiexec myprog infile1 : myprog infile2 : myprog infile3 47 48

**Example 11.15** Start 5 instances of the ocean program on x86\_64 hosts and 10 instances of the atmos program on Power9 hosts (Form B):

mpiexec -n 5 -arch x86\_64 ocean : -n 10 -arch power9 atmos

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

**Example 11.16** Start the ocean program on five Suns and the atmos program on 10 RS/6000's (Form B):

```
mpiexec -configfile myfile
```

where myfile contains

-n 5 -arch sun ocean -n 10 -arch rs6000 atmos

(End of advice to implementors.)

# 11.6 MPI and Threads

This section specifies the interaction between MPI calls and threads. Although thread compliance is not required, the standard specifies how threads are to work if they are provided. The section lists minimal requirements for **thread compliant** MPI implementations and defines functions that can be used for initializing the thread environment. MPI may be implemented in environments where threads are not supported or perform poorly. Therefore, MPI implementations are not required to be thread compliant as defined in this section. Regardless of whether or not the MPI implementation is thread compliant, a subset of MPI functions must always be thread safe. A complete list of such MPI functions is given in Table 11.1. When a thread is executing one of these routines, if another concurrently running thread also makes an MPI call, the outcome will be as if the calls executed in some order.

This section generally assumes a thread package similar to POSIX threads [44], but the syntax and semantics of thread calls are not specified here—these are beyond the scope of this document.

### 11.6.1 General

In a thread-compliant implementation, an MPI process is a process that may be multithreaded. Each thread can issue MPI calls; however, threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread. A message sent to a process can be received by any thread in this process.

*Rationale.* This model corresponds to the POSIX model of interprocess communication: the fact that a process is multithreaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations in which MPI 'processes' are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their "processes" are single-threaded). (*End of rationale.*) 1

2

3

4 5

6

7 8 9

10

11

12 13

14 15

16

17

18 19 20

21 22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35 36

37

38

39

40

41 42

43

44

45

46

Advice to users. It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread. (*End of advice to users.*)

- The two main requirements for a thread-compliant implementation are listed below.
- 1. All MPI calls are *thread-safe*, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.
- 2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. A blocked thread will not prevent *progress* of other runnable threads on the same process, and will not prevent them from executing MPI calls.

**Example 11.17** Process 0 consists of two threads. The first thread executes a blocking send call MPI\_Send(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI\_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

Advice to implementors. MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (End of advice to implementors.)

<sup>41</sup> 42 11.6.2 Clarifications

<sup>43</sup> Initialization and Completion When using the World Model, the call to MPI\_FINALIZE
 <sup>44</sup> should occur on the same thread that initialized MPI. We call this thread the main thread.
 <sup>45</sup> The call should occur only after all process threads have completed their MPI calls, and
 <sup>46</sup> have no pending communications or I/O operations.

 $^{24}$ 

Rationale. This constraint simplifies implementation. (End of rationale.)

Threads and the Sessions Model The Sessions Model provides a finer-grain approach to controlling the interaction between MPI calls and threads. When using this model, the desired level of thread support is specified at Session initialization time. See Section 11.3. Thus it is possible for communicators and other MPI objects derived from one Session to provide a different level of thread support than those created from another Session for which a different level of thread support was requested. Depending on the level of thread support requested at Session initialization time, different threads in a MPI process can make concurrent calls to MPI when using MPI objects derived from different *session handles.* Note that the requested and provided level of thread support when creating a Session may influence the granted level of thread support in a subsequent invocation of MPI\_SESSION\_INIT. Likewise, if the application at some point calls

MPI\_INIT\_THREAD, the requested and granted level of thread support may influence the granted level of thread support for subsequent calls to MPI\_SESSION\_INIT. Similarly, if the application calls MPI\_INIT\_THREAD after a call to MPI\_SESSION\_INIT, the level of thread support returned from MPI\_INIT\_THREAD may be similarly influenced by the requested level of thread support in the prior call to MPI\_SESSION\_INIT.

In addition, if an MPI application is only using the Sessions Model, the provided thread support level returned by MPI\_QUERY\_THREAD is the same as that returned prior to invocation of MPI\_INIT\_THREAD or MPI\_INIT. If the application also used the World Model in some component of the application, MPI\_QUERY\_THREAD will return the level of thread support returned by the original call to MPI\_INIT\_THREAD.

Multiple threads completing the same request. A program in which two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent MPI\_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test that violates this rule is erroneous.

Rationale. This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI\_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an MPI\_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (End of rationale.)

Probe A receive call that uses source and tag values returned by a preceding call to MPI\_PROBE or MPI\_IPROBE will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multi-threaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process. Alternatively, MPI\_MPROBE or MPI\_IMPROBE can be used.

**Collective calls** Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{24}$ 

 $45 \\ 46$ 

# 520 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

threads issue such calls on the same communicator, window or file handle, it is up to the
 user to make sure the calls are correctly ordered, using interthread synchronization.

Advice to users. With three concurrent threads in each MPI process of a communicator comm, it is allowed that thread A in each MPI process calls a collective operation on comm, thread B calls a file operation on an existing file handle that was formerly opened on comm, and thread C invokes one-sided operations on an existing window handle that was also formerly created on comm. (*End of advice to users.*)

*Rationale.* As specified in MPI\_FILE\_OPEN and MPI\_WIN\_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

Advice to implementors. If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (End of advice to implementors.)

<sup>18</sup> <sup>19</sup> Error handlers An error handler does not necessarily execute in the context of the thread <sup>20</sup> that made the error-raising MPI call; the error handler may be executed by a thread that <sup>21</sup> is distinct from the thread that will return the error code.

*Rationale.* The MPI implementation may be multithreaded, so that part of the communication protocol may execute on a thread that is distinct from the thread that made the MPI call. The design allows the error handler to be executed on the thread where the error is raised. (*End of rationale.*)

Interaction with signals and cancellations The outcome is undefined if a thread that executes
 an MPI call is cancelled (by another thread), or if a thread catches a signal while executing
 an MPI call. However, a thread of an MPI process may terminate, and may catch signals or
 be cancelled by another thread when not executing MPI calls.

*Rationale.* Few C library functions are signal safe, and many have cancellation points—points at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe"). (*End of rationale.*)

Advice to users. Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to sigwait for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

Advice to implementors. The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

# 11.7 The Dynamic Process Model

The dynamic process model allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not "start" the other.

### 11.7.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager.

MPI\_COMM\_SPAWN starts MPI processes and establishes communication with them, returning an inter-communicator. MPI\_COMM\_SPAWN\_MULTIPLE starts several different binaries (or the same binary with different arguments), placing them in the same MPI\_COMM\_WORLD and returning an inter-communicator.

MPI uses the group abstraction to represent processes. A process is identified by a (group, rank) pair.

11.7.2 The Runtime Environment

The MPI\_COMM\_SPAWN and MPI\_COMM\_SPAWN\_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one.

MPI assumes, implicitly, the existence of an environment in which an application runs. It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes, to find out properties of the runtime environment (how many processors, how much memory, etc.). Complex interaction of an MPI application with its runtime environment should be done through an environmentspecific API.

At some low level, obviously, MPI must be able to interact with the runtime system, but the interaction is not visible at the application level and the details of the interaction are not specified by the MPI standard.

In many cases, it is impossible to keep environment-specific information out of the MPI interface without seriously compromising MPI functionality. To permit applications to take advantage of environment-specific functionality, many MPI routines take an info argument that allows an application to specify environment-specific information. There is a tradeoff between functionality and portability: applications that make use of environment-specific info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

 $^{24}$ 

- A process may start new processes with MPI\_COMM\_SPAWN and MPI\_COMM\_SPAWN\_MULTIPLE.
  - When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.
- An attribute MPI\_UNIVERSE\_SIZE (See Section 11.10.1) on MPI\_COMM\_WORLD tells a program how "large" the initial runtime environment is, namely how many processes can usefully be started in all. One can subtract the size of MPI\_COMM\_WORLD from this value to find out how many processes might usefully be started in addition to those already running.

 $\mathbf{2}$ 

# 11.8 Process Manager Interface

11.8.1 Processes in MPI

<sup>17</sup> A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a unique process but a process does not determine a unique (group, rank) pair, since a process may belong to several groups.

# 11.8.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with
 them, returning an inter-communicator.

Advice to users. It is possible in MPI to start an SPMD or MPMD application with a fixed number of processes after initialization by first starting one process and having that process start its siblings with MPI\_COMM\_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI application. (*End of advice to users.*)

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, <sup>1</sup> array_of_errcodes) <sup>2</sup>				
	- ,		3	
IN	command	name of program to be spawned (string, significant	4	
		only at root)	5	
IN	argv	arguments to $command$ (array of strings, significant	6	
		only at root)	7	
IN	maxprocs	maximum number of processes to start (integer,	8	
		significant only at root)	9	
IN	info	a set of key-value pairs telling the runtime system	10	
		where and how to start the processes (handle,	11	
		significant only at root)	12	
IN	root	rank of process in which previous arguments are	13	
IIN	1001	examined (integer)	14 15	
IN	comm	intra-communicator containing group of spawning	16	
processes (handle)		processes (handle)	17	
OUT	intercomm	inter-communicator between original group and the	18	
		newly spawned group (handle)	19	
OUT	array_of_errcodes	one code per process (array of integers)	20	
001	anay_or_encodes	one code per process (array of integers)	21	
			22	
C bindir	0	accommond above to your [] int mourneed	23 24	
int MPI_	-	<pre>command, char *argv[], int maxprocs, t root, MPI_Comm comm, MPI_Comm *intercomm,</pre>	24 25	
	int array_of_errco		26	
	int array_or_erred		27	
	2008 binding		28	
MPI_Comm		axprocs, info, root, comm, intercomm,	29	
	array_of_errcodes		30	
	ACTER(LEN=*), INTENT(IN	6	31	
	GER, INTENT(IN) :: maxp		32	
TYPE(MPI_Info), INTENT(IN) :: info				
	C(MPI_Comm), INTENT(IN)		34	
TYPE(MPI_Comm), INTENT(OUT) :: intercomm INTEGER :: array_of_errcodes(*)				
INTEGED ODTIONAL INTENT(OUT) · · jorror				

ARRAY\_OF\_ERRCODES, IERROR) CHARACTER\*(\*) COMMAND, ARGV(\*) INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY\_OF\_ERRCODES(\*), IERROR MPI\_COMM\_SPAWN tries to start maxprocs identical copies of the MPI program spec-

MPI\_COMM\_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding

45ified by command, establishing communication with them and returning an inter-commu-46nicator. The spawned processes are referred to as children. The children have their own 47MPI\_COMM\_WORLD, which is separate from that of the parents. MPI\_COMM\_SPAWN is

37

38

39

40 $^{41}$ 

42

43 44

 $^{48}$ 

# 524 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 collective over comm, and also may not return until MPI\_INIT has been called in the chil- $\mathbf{2}$ dren. Similarly, MPI\_INIT in the children may not return until all parents have called 3 MPI\_COMM\_SPAWN. In this sense, MPI\_COMM\_SPAWN in the parents and MPI\_INIT in 4 the children form a collective operation over the union of parent and child processes. The  $\mathbf{5}$ inter-communicator returned by MPI\_COMM\_SPAWN contains the parent processes in the 6 local group and the child processes in the remote group. The ordering of processes in the  $\overline{7}$ local and remote groups is the same as the ordering of the group of the comm in the parents 8 and of MPI\_COMM\_WORLD of the children, respectively. This inter-communicator can be 9 obtained in the children through the function MPI\_COMM\_GET\_PARENT.

10 11

12

13

14

15 16 Advice to users. An implementation may automatically establish communication before MPI\_INIT is called by the children. Thus, completion of MPI\_COMM\_SPAWN in the parent does not necessarily mean that MPI\_INIT has been called in the children (although the returned inter-communicator can be used immediately). (*End of advice to users.*)

The command argument The command argument is a string containing the name of a pro gram to be spawned. The string is null-terminated in C. In Fortran, leading and trailing
 spaces are stripped. MPI does not specify how to find the executable or how the working
 directory is determined. These rules are implementation-dependent and should be appro priate for the runtime environment.

22 23

24

25

26

27

28

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation should document its rules for finding executables and determining working directories, and a high-quality implementation should give the user some control over these rules. (End of advice to implementors.)

29 30 31

32

33 34

35

36

37

38

39

If the program named in **command** does not call MPI\_INIT, but instead forks a process that calls MPI\_INIT, the results are undefined. Implementations may allow this case to work but are not required to.

Advice to users. MPI does not say what happens if the program you start is a shell script and that shell script starts a program that calls MPI\_INIT. Though some implementations may allow you to do this, they may also have restrictions, such as requiring that arguments supplied to the shell script be supplied to the program, or requiring that certain parts of the environment not be changed. (*End of advice to users.*)

40 41

The argv argument argv is an array of strings containing arguments that are passed to the program. The first element of argv is the first argument passed to command, not, as is conventional in some contexts, the command itself. The argument list is terminated by NULL in C and an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI\_ARGV\_NULL may be used in C and Fortran to indicate an empty argument list. In C this constant is the same as NULL.

```
Example 11.18 Examples of argv in C and Fortran
To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" in C:
       char command[] = "ocean";
       char *argv[] = {"-gridfile", "ocean1.grd", NULL};
       MPI_Comm_spawn(command, argv, ...);
or, if not everything is known at compile time:
       char *command;
       char **argv;
       command = "ocean";
       argv=(char **)malloc(3 * sizeof(char *));
       argv[0] = "-gridfile";
       argv[1] = "ocean1.grd";
       argv[2] = NULL;
       MPI_Comm_spawn(command, argv, ...);
In Fortran:
       CHARACTER*25 command, argv(3)
       command = 'ocean'
       argv(1) = '-gridfile'
       argv(2) = 'ocean1.grd'
       argv(3) = ', '
       call MPI_COMM_SPAWN(command, argv, ...)
```

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI\_COMM\_SPAWN argument argv differs from the argv argument of main in two respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the implementation and conventionally contains the name of the program (given by command). argv[1] of main corresponds to argv[0] in MPI\_COMM\_SPAWN, argv[2] of main to argv[1] of MPI\_COMM\_SPAWN, etc. Passing an argv of MPI\_ARGV\_NULL to MPI\_COMM\_SPAWN results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program. Second, argv of MPI\_COMM\_SPAWN must be null-terminated, so that its length can be determined.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in argv of main(), the MPI implementation may add the arguments to the argv that is passed to MPI\_INIT.

The maxprocs argument MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI\_ERR\_SPAWN.

An implementation may allow the info argument to change the default behavior, such that if the implementation is unable to spawn all maxprocs processes, it may spawn a smaller number of processes instead of raising an error. In principle, the info argument may specify an arbitrary set  $\{m_i : 0 \le m_i \le \text{maxprocs}\}$  of allowed values for the number of processes spawned. The set  $\{m_i\}$  does not necessarily include the value maxprocs. If an implementation is able to spawn one of these allowed numbers of processes,

1

 $\mathbf{2}$ 

3

4

5

6 7

8 9

10

11

12

13

14

15

16

17

18 19

20

21

22

23

24

25 26

27

28

29

30

31

32

33

34

35

36

37

38

39 40

41

42

43

44

45

# 526 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

<sup>1</sup> MPI\_COMM\_SPAWN returns successfully and the number of spawned processes, *m*, is given <sup>2</sup> by the size of the remote group of intercomm. If *m* is less than maxproc, reasons why the <sup>3</sup> other processes were not spawned are given in array\_of\_errcodes as described below. If it is <sup>4</sup> not possible to spawn one of the allowed numbers of processes, MPI\_COMM\_SPAWN raises <sup>5</sup> an error of class MPI\_ERR\_SPAWN.

A spawn call with the default behavior is called *hard*. A spawn call for which fewer than
 maxprocs processes may be returned is called "soft". See Section 11.8.4 for more information
 on the "soft" key for info.

- Advice to users. By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to N," you should do a soft spawn, where the set of allowed values  $\{m_i\}$  is  $\{0, \ldots, N\}$ . However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)
- 15 16

9

10

11

12

13

14

<sup>17</sup> The info argument The info argument to all of the routines in this chapter is an opaque han-<sup>18</sup> dle of type MPI\_Info in C and Fortran with the mpi\_f08 module and INTEGER in Fortran with <sup>19</sup> the mpi module or the include file mpif.h. It is a container for a number of user-specified <sup>20</sup> (key,value) pairs. key and value are strings (null-terminated char\* in C, character\*(\*) in <sup>21</sup> Fortran). Routines to create and manipulate the info argument are described in Chapter 10.

For the SPAWN calls, info provides additional (and possibly implementation-dependent)
 instructions to MPI and the runtime system on how to start processes. An application may
 pass MPI\_INFO\_NULL in C or Fortran. Portable programs not requiring detailed control over
 process locations should use MPI\_INFO\_NULL.

<sup>26</sup> MPI does not specify the content of the info argument, except to reserve a number of <sup>27</sup> special key values (see Section 11.8.4). The info argument is quite flexible and could even <sup>28</sup> be used, for example, to specify the executable and its command-line arguments. In this <sup>29</sup> case the command argument to MPI\_COMM\_SPAWN could be empty. The ability to do this <sup>30</sup> follows from the fact that MPI does not specify how an executable is found, and the info <sup>31</sup> argument can tell the runtime system where to "find" the executable "" (empty string). Of <sup>32</sup> course, a program that does this will not be portable across MPI implementations.

33

The root argument All arguments before the root argument are examined only on the process whose rank in comm is equal to root. The value of these arguments on other processes is ignored.

37

38 The array\_of\_errcodes argument The array\_of\_errcodes is an array of length maxprocs in 39 which MPI reports the status of each process that MPI was requested to start. If all maxprocs 40processes were spawned,  $\operatorname{array_of}$  errcodes is filled in with the value MPI\_SUCCESS. If only m 41  $(0 \le m \le maxprocs)$  processes are spawned, m of the entries will contain MPI\_SUCCESS and 42the rest will contain an implementation-specific error code indicating the reason MPI could 43 not start the process. MPI does not specify which entries correspond to failed processes. 44An implementation may, for instance, fill in error codes in one-to-one correspondence with 45a detailed specification in the info argument. These error codes all belong to the error class 46MPI\_ERR\_SPAWN if there was no error in the argument list. In C or Fortran, an application 47may pass MPI\_ERRCODES\_IGNORE if it is not interested in the error codes. 48

Advice to implementors. MPI\_ERRCODES\_IGNORE in Fortran is a special type of constant, like MPI\_BOTTOM. See the discussion in Section 2.5.4. (End of advice to implementors.)

MPI_COMM_GET_PARENT(parent)					
OUTparentthe parent communicator (handle)					
C binding int MPI_Comm_get_parent(MPI_Comm *parent)					
Fortran 2008 binding MPI_Comm_get_parent(parent, ierror) TYPE(MPI_Comm), INTENT(OUT) :: parent INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
Fortran binding MPI_COMM_GET_PARENT(PARENT, IERROR) INTEGER PARENT, IERROR					
If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, MPI_COMM_GET_PARENT returns the "parent" inter-communicator of the current pro- cess. This parent inter-communicator is created implicitly inside of MPI_INIT and is the same inter-communicator returned by SPAWN in the parents. If the process was not spawned, MPI_COMM_GET_PARENT returns MPI_COMM_NULL. After the parent communicator is freed or disconnected, MPI_COMM_GET_PARENT returns MPI_COMM_NULL.					
Advice to users. MPI_COMM_GET_PARENT returns a handle to a single inter- communicator. Calling MPI_COMM_GET_PARENT a second time returns a handle to the same inter-communicator. Freeing the handle with MPI_COMM_DISCONNECT or MPI_COMM_FREE will cause other references to the inter-communicator to become invalid (dangling). Note that calling MPI_COMM_FREE on the parent communicator is not useful. ( <i>End of advice to users.</i> )					
<i>Rationale.</i> The desire of the Forum was to create a constant MPI_COMM_PARENT similar to MPI_COMM_WORLD. Unfortunately such a constant cannot be used (syntactically) as an argument to MPI_COMM_DISCONNECT, which is explicitly allowed. ( <i>End of rationale.</i> )					
11.8.3 Starting Multiple Executables and Establishing Communication					
While MPI_COMM_SPAWN is sufficient for most cases, it does not allow the spawning of multiple binaries, or of the same binary with multiple sets of arguments. The following routine spawns multiple binaries or the same binary with multiple sets of arguments, establishing communication with them and placing them in the same MPI_COMM_WORLD.					

 $\mathbf{2}$ 

 $^{41}$ 

```
    MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv,
array_of_maxprocs, array_of_info, root, comm, intercomm,
array_of_errcodes)
```

3		array_of_errcodes)	y_or_inio, root, comm, intercomm,		
4 5 6	IN	count	number of commands (positive integer, significant only at root)		
7 8	IN	array_of_commands	programs to be executed (array of strings, significant only at root)		
9 10	IN	array_of_argv	arguments for <b>commands</b> (array of array of strings, significant only at root)		
11 12	IN	array_of_maxprocs	maximum number of processes to start for each command (array of integers, significant only at root)		
13 14 15 16	IN	array_of_info	info objects telling the runtime system where and how to start processes (array of handles, significant only at root)		
17 18	IN	root	rank of process in which previous arguments are examined (integer)		
19 20 21	IN	comm	intra-communicator containing group of spawning processes (handle)		
22 23	OUT	intercomm	inter-communicator between original group and the newly spawned group (handle)		
24 25	OUT	array_of_errcodes	one error code per process (array of integers)		
26 27 28 29 30 31		omm_spawn_multiple(int c char **array_of_argv const MPI_Info array MPI_Comm *intercomm,	<pre>ount, char *array_of_commands[], v[], const int array_of_maxprocs[], v_of_info[], int root, MPI_Comm comm, v_ int array_of_errcodes[])</pre>		
32 33 34			ray_of_commands, array_of_argv, array_of_info, root, comm, intercomm,		
35 36 37 38 39	<pre>array_of_errcodes, ierror) INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),</pre>				
40 41 42 43	TYPE(MPI_Comm), INTENT(IN) :: allay_Of_INIO(*) TYPE(MPI_Comm), INTENT(IN) :: comm INTEGER :: array_of_errcodes(*) INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
44 45 46 47 48	Fortran binding MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV, ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)				

INTEGER COUNT, ARRAY\_OF\_MAXPROCS(\*), ARRAY\_OF\_INFO(\*), ROOT, COMM, INTERCOMM, ARRAY\_OF\_ERRCODES(\*), IERROR CHARACTER\*(\*) ARRAY\_OF\_COMMANDS(\*), ARRAY\_OF\_ARGV(COUNT, \*)

MPI\_COMM\_SPAWN\_MULTIPLE is identical to MPI\_COMM\_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI\_COMM\_SPAWN. For the Fortran version of array\_of\_argv, the element array\_of\_argv(i,j) is the j-th argument to command number i.

*Rationale.* This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI\_COMM\_SPAWN to sort out arguments. Note that the leading dimension of array\_of\_argv must be the same as count. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI\_COMM\_SPAWN\_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to count. This Fortran feature allows an implementor to define MPI\_ARGVS\_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (End of rationale.)

Advice to users. The argument count is interpreted by MPI only at the root, as is array\_of\_argv. Since the leading dimension of array\_of\_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array\_of\_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (End of advice to users.)

In any language, an application may use the constant MPI\_ARGVS\_NULL (which is likely to be (char \*\*\*)0 in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array\_of\_argv to MPI\_ARGV\_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding argv whose first element is null ((char \*)0 in C and empty string in Fortran). In Fortran at non-root processes, the count argument must be set to a value that is consistent with the provided array\_of\_argv although the content of these arguments has no meaning for this operation.

All of the spawned processes have the same MPI\_COMM\_WORLD. Their ranks in MPI\_COMM\_WORLD correspond directly to the order in which the commands are specified in MPI\_COMM\_SPAWN\_MULTIPLE. Assume that  $m_1$  processes are generated by the first command,  $m_2$  by the second, etc. The processes corresponding to the first command have ranks  $0, 1, \ldots, m_1-1$ . The processes in the second command have ranks  $m_1, m_1+1, \ldots, m_1+m_2-1$ . The processes in the third have ranks  $m_1 + m_2, m_1 + m_2 + 1, \ldots, m_1 + m_2 + m_3 - 1$ , etc.

Advice to users. Calling MPI\_COMM\_SPAWN multiple times would create many sets of children with different MPI\_COMM\_WORLDs whereas

MPI\_COMM\_SPAWN\_MULTIPLE creates children with a single MPI\_COMM\_WORLD, 45 so the two methods are not completely equivalent. There are also two performancerelated reasons why, if you need to spawn multiple executables, you may want to use MPI\_COMM\_SPAWN\_MULTIPLE instead of calling MPI\_COMM\_SPAWN several 48

 $\overline{7}$ 

 $^{24}$ 

times. First, spawning several things at once may be faster than spawning them sequentially. Second, in some implementations, communication between processes spawned at the same time may be faster than communication between processes spawned separately. (End of advice to users.)

The array\_of\_errcodes argument is a 1-dimensional array of size  $\sum_{i=1}^{count} n_i$ , where  $n_i$  is the *i*-th element of array\_of\_maxprocs. Command number *i* corresponds to the  $n_i$  contiguous slots in this array from element  $\sum_{j=1}^{i-1} n_j$  to  $\left[\sum_{j=1}^{i} n_j\right] - 1$ . Error codes are treated the same 8 9 as with MPI\_COMM\_SPAWN. 10

```
Example 11.19 Examples of array_of_argv in C and Fortran
11
     To run the program "ocean" with arguments "-gridfile" and "ocean1.grd" and the program
12
     "atmos" with argument "atmos.grd" in C:
13
14
             char *array_of_commands[2] = {"ocean", "atmos"};
15
             char **array_of_argv[2];
16
             char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
17
             char *argv1[] = {"atmos.grd", (char *)0};
18
             array_of_argv[0] = argv0;
19
             array_of_argv[1] = argv1;
20
             MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
21
22
     Here is how you do it in Fortran:
23
             CHARACTER*25 commands(2), array_of_argv(2, 3)
^{24}
             commands(1) = 'ocean'
25
             array_of_argv(1, 1) = '-gridfile'
26
             array_of_argv(1, 2) = 'ocean1.grd'
27
             array_of_argv(1, 3) = ', '
28
29
             commands(2) = 'atmos'
30
             array_of_argv(2, 1) = 'atmos.grd'
31
             array_of_argv(2, 2) = ', '
32
33
             call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
34
```

# 11.8.4 Reserved Keys

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

35 36

37

38

3940

41

4243

44

45

The following keys are reserved. An implementation is not required to interpret these keys, but if it does interpret the key, it must provide the functionality described.

"host" Value is a hostname. The format of the hostname is determined by the implementation.

"arch" Value is an architecture name. Valid architecture names and what they mean are determined by the implementation.

"wdir" Value is the name of a directory on a machine on which the spawned process(es) 46execute(s). This directory is made the working directory of the executing process(es). 47 The format of the directory name is determined by the implementation. 48

- "path" Value is a directory or set of directories where the implementation should look for the executable. The format of "path" is determined by the implementation.
- "file" Value is the name of a file in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.
- "mpi\_initial\_errhandler" Value is the name of an errhandler that will be set as the initial error handler. The "mpi\_initial\_errhandler" key can take the case insensitive values "mpi\_errors\_are\_fatal", "mpi\_errors\_abort", and "mpi\_errors\_return" representing the predefined MPI error handlers (MPI\_ERRORS\_ARE\_FATAL—the default, MPI\_ERRORS\_ABORT, and MPI\_ERRORS\_RETURN, respectively). Other, nonstandard values may be supported by the implementation, which should document the resultant behavior.
- "soft" Value specifies a set of numbers which are allowed values for the number of processes 14that MPI\_COMM\_SPAWN (et al.) may create. The format of the value is a commaseparated list of Fortran-90 triplets each of which specifies a set of integers and which together specify the set formed by the union of these sets. Negative values in this set and values greater than maxprocs are ignored. MPI will spawn the largest number of processes it can, consistent with some number in the set. The order in which triplets are given is not significant.

By Fortran-90 triplets, we mean:

- 1. a means a
- 2. a:b means a, a + 1, a + 2, ..., b
- 3. a:b:c means  $a, a + c, a + 2c, \ldots, a + ck$ , where for c > 0, k is the largest integer for which  $a + ck \le b$  and for c < 0, k is the largest integer for which  $a + ck \ge b$ . If b > a then c must be positive. If b < a then c must be negative.

Examples:

- 1. **a:b** gives a range between a and b
- 2. 0:N gives full "soft" functionality
- 3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two number of processes.
- 4. 2:10000:2 allows an even number of processes up to a maximum of 10,000 processes.
- 5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.

```
11.8.5 Spawn Example
```

**Example 11.20** Manager-worker Example Using MPI\_COMM\_SPAWN

```
/* manager */
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
```

1

2

3

4

56

 $\overline{7}$ 

8

9

10

11

12

13

15

16

17

18

19

2021

22

23

 $^{24}$ 25

26

27

28

2930

31

32

33

34 35

36

37

38 39

40 41

4243

44

45

46

47

```
1
        int world_size, universe_size, *universe_sizep, flag;
\mathbf{2}
        MPI_Comm everyone;
                                       /* inter-communicator */
3
        char worker_program[100];
4
5
        MPI_Init(&argc, &argv);
6
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
7
8
        if (world_size != 1)
                                  error("Top heavy with management");
9
10
        MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
11
                           &universe_sizep, &flag);
12
        if (!flag) {
13
             printf("This MPI does not support UNIVERSE_SIZE. How many\n\
14
     processes total?");
15
              scanf("%d", &universe_size);
16
        } else universe_size = *universe_sizep;
17
        if (universe_size == 1) error("No room to start workers");
18
19
        /*
20
         * Now spawn the workers. Note that there is a run-time determination
21
         * of what type of worker to spawn, and presumably this calculation must
22
         * be done at run time and cannot be calculated before starting
23
         * the program. If everything is known when the application is
24
         * first started, it is generally better to start them all at once
25
         * in a single MPI_COMM_WORLD.
26
         */
27
28
        choose_worker_program(worker_program);
29
        MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
30
                   MPI_INFO_NULL, 0, MPI_COMM_SELF, & everyone,
31
                   MPI_ERRCODES_IGNORE);
32
        /*
33
         * Parallel code here. The communicator "everyone" can be used
34
         * to communicate with the spawned processes, which have ranks 0,...
35
         * MPI_UNIVERSE_SIZE-1 in the remote group of the inter-communicator
36
         * "everyone".
37
         */
38
39
        MPI_Finalize();
40
        return 0;
41
     }
42
43
     /* worker */
44
45
     #include "mpi.h"
46
     int main(int argc, char *argv[])
47
     {
48
```

```
int size;
MPI_Comm parent;
MPI_Init(&argc, &argv);
MPI_Comm_get_parent(&parent);
if (parent == MPI_COMM_NULL) error("No parent!");
MPI_Comm_remote_size(parent, &size);
if (size != 1) error("Something's wrong with the parent");
/*
 * Parallel code here.
 * The manager is represented as the process with rank 0 in (the remote
 * group of) the parent communicator. If the workers need to communicate
 * among themselves, they can use MPI_COMM_WORLD.
 */
MPI_Finalize();
return 0;
```

# 11.9 Establishing Communication

}

This section provides functions that establish communication between two sets of MPI processes that do not share a communicator.

Some situations in which these functions are useful are:

- 1. Two parts of an application that are started independently need to communicate.
- 2. A visualization tool wants to attach to a running process.
- 3. A server wants to accept connections from multiple clients. Both clients and server may be parallel programs.

In each of these situations, MPI must establish communication channels where none existed before, and there is no parent/child relationship. The routines described in this section establish communication between the two sets of processes by creating an MPI inter-communicator, where the two groups of the inter-communicator are the original sets of processes.

Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. The other group connects to the server; we will call it the *client*.

Advice to users.While the names client and server are used throughout this section,43MPI does not guarantee the traditional robustness of client/server systems.The func-tionality described in this section is intended to allow two cooperating parts of the45same application to communicate with one another.For instance, a client that gets asegmentation fault and dies, or one that does not participate in a collective operation47may cause a server to crash or hang.(End of advice to users.)48

1

2

4

5

6

7

8 9

10

11

12

13

14

15 16

17

18

19 20

21 22

23

24

25 26

27

28 29

30

31 32

33

34

35

36

37

38

39

40

41

#### 11.9.1 Names, Addresses, Ports, and All That

1

7

8

9

11

1213

14

15

1617

18

19

20

2122

23

24

25

26

27

28

29

30

 $^{31}$ 

32

33

39

40 41

42

43

44

45

 $\mathbf{2}$ Almost all of the complexity in MPI client/server routines addresses the question "how 3 does the client find out how to contact the server?" The difficulty, of course, is that there 4 is no existing communication channel between them, yet they must somehow agree on a 5rendezvous point where they will establish communication. 6

Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client does not really care what server it contacts, only that it be able to get in touch with one that can handle its request. 10

Ideally, MPI can accommodate a wide variety of run-time systems while retaining the ability to write simple, portable code. The following should be compatible with MPI:

- The server resides at a well-known internet address host:port.
- The server prints out an address to the terminal; the user gives this address to the client program.
  - The server places the address information on a nameserver, where it can be retrieved with an agreed-upon name.
  - The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.

MPI does not require a nameserver, so not all implementations will be able to support all of the above scenarios. However, MPI provides an optional nameserver interface, and is compatible with external name servers.

A port\_name is a system-supplied string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol. The server establishes a port\_name with the MPI\_OPEN\_PORT routine. It accepts a connection to a given port with MPI\_COMM\_ACCEPT. A client uses port\_name to connect to the server.

By itself, the port\_name mechanism is completely portable, but it may be clumsy to use because of the necessity to communicate port\_name to the client. It would be more convenient if a server could specify that it be known by an *application-supplied* service\_name so that the client could connect to that service\_name without knowing the port\_name.

34 An MPI implementation may allow the server to publish a (port\_name, service\_name) 35 pair with MPI\_PUBLISH\_NAME and the client to retrieve the port name from the service 36 name with MPI\_LOOKUP\_NAME. This allows three levels of portability, with increasing 37 levels of functionality. 38

- 1. Applications that do not rely on the ability to publish names are the most portable. Typically the port\_name must be transferred "by hand" from server to client.
- 2. Applications that use the MPI\_PUBLISH\_NAME mechanism are completely portable among implementations that provide this service. To be portable among all implementations, these applications should have a fall-back mechanism that can be used when names are not published.
- 463. Applications may ignore MPI's name publishing functionality and use their own mech-47anism (possibly system-supplied) to publish names. This allows arbitrary flexibility 48 but is not portable.

11.9.2 Server Routines			1
A server makes itself available with two routines. First it must call MPI_OPEN_PORT to establish a <b>port</b> at which it may be contacted. Secondly it must call MPI_COMM_ACCEPT to accept connections from clients.			2
			3
			4
to accept	connections from chefts.		5
			6
MPI_OPEN_PORT(info, port_name)			7
IN	info	implementation-specific information on how to	8
	inio	establish an address (handle)	9
			10
OUT	port_name	newly established port (string)	11
			12
C binding			13
<pre>int MPI_Open_port(MPI_Info info, char *port_name)</pre>			14
<pre>Fortran 2008 binding MPI_Open_port(info, port_name, ierror)    TYPE(MPI_Info), INTENT(IN) :: info    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name    INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>			15
			16 17
			18
			19
			20
			21
Fortran binding			22
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR) INTEGER INFO, IERROR CHARACTER*(*) PORT_NAME			23
			24
			25
This function establishes a network address, encoded in the port_name string, at which the server will be able to accept connections from clients. port_name is supplied by the system, possibly using information in the info argument. MPI copies a system-supplied port name into port_name. port_name identifies the newly opened port and can be used by a client to contact the server. The maximum size string that may be supplied by the system is MPI_MAX_PORT_NAME.			26
			27
			28
			29
			30
			31
Advice to users. The system copies the port name into port_name. The application must pass a buffer of sufficient size to hold this value. ( <i>End of advice to users.</i> )			32
			33
			34
port_name is essentially a network address. It is unique within the communication universe to which it belongs (determined by the implementation), and may be used by any client within that communication universe. For instance, if it is an internet (host:port) address, it will be unique on the internet. If it is a low level switch address on an IBM SP, it will be unique to that SP.			35
			36
			37
			38
			39

Advice to implementors. These examples are not meant to constrain implementations. A port\_name could, for instance, contain a user name or the name of a batch job, as long as it is unique within some well-defined communication domain. The larger the communication domain, the more useful MPI's client/server functionality will be. (End of advice to implementors.)

The precise form of the address is implementation-defined. For instance, an internet address may be a host name or IP address, or anything that the implementation can decode into

40

41

42

43

44

4546

47

1 an IP address. A port name may be reused after it is freed with MPI\_CLOSE\_PORT and  $\mathbf{2}$ released by the system. 3 4 Advice to implementors. Since the user may type in port\_name by hand, it is useful to choose a form that is easily readable and does not have embedded spaces. (End of 5advice to implementors.) 6 7 info may be used to tell the implementation how to establish the address. It may, and 8 usually will, be MPI\_INFO\_NULL in order to get the implementation defaults. 9 10 11 MPI\_CLOSE\_PORT(port\_name) 1213 IN port\_name a port (string) 1415C binding 16int MPI\_Close\_port(const char \*port\_name) 17Fortran 2008 binding 18 MPI\_Close\_port(port\_name, ierror) 19 CHARACTER(LEN=\*), INTENT(IN) :: port\_name 20INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2122Fortran binding 23MPI\_CLOSE\_PORT(PORT\_NAME, IERROR)  $^{24}$ CHARACTER\*(\*) PORT\_NAME 25INTEGER IERROR 26This function releases the network address represented by port\_name. 272829MPI\_COMM\_ACCEPT(port\_name, info, root, comm, newcomm) 30 IN port name (string, significant only at root) port\_name  $^{31}$ 32 IN info implementation-dependent information (handle, 33 significant only at root) 34IN rank in comm of root node (integer) root 35 intra-communicator over which call is collective IN comm 36 (handle) 37 38OUT inter-communicator with client as remote group newcomm 39 (handle) 40 $^{41}$ C binding 42int MPI\_Comm\_accept(const char \*port\_name, MPI\_Info info, int root, 43 MPI\_Comm comm, MPI\_Comm \*newcomm) 44 Fortran 2008 binding 45MPI\_Comm\_accept(port\_name, info, root, comm, newcomm, ierror) 46 CHARACTER(LEN=\*), INTENT(IN) :: port\_name 47TYPE(MPI\_Info), INTENT(IN) :: info 48

	EGER, INTENT(IN) ::		1	
	E(MPI_Comm), INTENT		2 3	
	E(MPI_Comm), INTENT EGER, OPTIONAL, INT		3 4	
			5	
	binding		6	
	M_ACCEPT(PORT_NAME, RACTER*(*) PORT_NAM	INFO, ROOT, COMM, NEWCOMM, IERROR)	7	
	· · · —	E MM, NEWCOMM, IERROR	8	
			9	
		ablishes communication with a client. It is collective over the	10 11	
the clien		ns an inter-communicator that allows communication with	11	
		been established through a call to MPI_OPEN_PORT.	13	
	•	e directives that may influence the behavior of the ACCEPT	14	
call.	I I I I I I I I I I I I I I I I I I I		15	
			16	
11.9.3	Client Routines		17	
There is	only one routine on th	e client side	18 19	
1 11010 10	only one routine on th		20	
			21	
MPI_CO	MM_CONNECT(port_r	name, info, root, comm, newcomm)	22	
IN	port_name	network address (string, significant only at root)	23	
IN	info	implementation-dependent information (handle,	24	
		significant only at root)	25	
IN	root	rank in comm of root node (integer)	26 27	
IN	comm	intra-communicator over which call is collective	28	
		(handle)	29	
OUT	newcomm	inter-communicator with server as remote group	30	
		(handle)	31	
			32	
C bindi	ng		33 34	
int MPI		char *port_name, MPI_Info info, int root,	34	
	MPI_Comm comm	n, MPI_Comm *newcomm)	36	
Fortran	2008 binding		37	
MPI_Com	m_connect(port_name	e, info, root, comm, newcomm, ierror)	38	
		NT(IN) :: port_name	39	
	TYPE(MPI_Info), INTENT(IN) :: info INTEGER, INTENT(IN) :: root TYPE(MPI_Comm), INTENT(IN) :: comm			
	TYPE(MPI_Comm), INTENT(UN) :: newcomm			
	EGER, OPTIONAL, INT		43 44	
			45	
	Fortran binding MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)			
	CHARACTER*(*) PORT_NAME			

INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

This routine establishes communication with a server specified by port\_name. It is collective over the calling communicator and returns an inter-communicator in which the remote group participated in an MPI\_COMM\_ACCEPT.

If the named port does not exist (or has been closed), MPI\_COMM\_CONNECT raises an error of class MPI\_ERR\_PORT.

If the port exists, but does not have a pending MPI\_COMM\_ACCEPT, the connection attempt will eventually time out after an implementation-defined time, or succeed when the server calls MPI\_COMM\_ACCEPT. In the case of a time out, MPI\_COMM\_CONNECT raises an error of class MPI\_ERR\_PORT.

However, a high-quality implementation will try to queue connection attempts so

that a server can handle simultaneous requests from several clients. A high-quality

implementation may also provide a mechanism, through the info arguments to

The time out period may be arbitrarily short or long.

11 12

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

- $13 \\ 14$
- 15
- 16

17

19

20

21

22

23

 $^{24}$ 

25

26

18

MPI\_OPEN\_PORT, MPI\_COMM\_ACCEPT, and/or MPI\_COMM\_CONNECT, for the user to control timeout and queuing behavior. (*End of advice to implementors.*)

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order they were initiated and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

port\_name is the address of the server. It must be the same as the name returned by MPI\_OPEN\_PORT on the server. Some freedom is allowed here. If there are equivalent forms of port\_name, an implementation may accept them as well. For instance, if port\_name is (hostname:port), an implementation may accept (ip\_address:port) as well.

27 28 29

# 11.9.4 Name Publishing

Advice to implementors.

The routines in this section provide a mechanism for publishing names. A (service\_name, 30  $^{31}$ port\_name) pair is published by the server, and may be retrieved by a client using the 32 service\_name only. An MPI implementation defines the scope of the service\_name, that 33 is, the domain over which the service\_name can be retrieved. If the domain is the empty 34set, that is, if no client can retrieve the information, then we say that name publishing 35 is not supported. Implementations should document how the scope is determined. High-36 quality implementations will give some control to users through the info arguments to name 37 publishing functions. Examples are given in the descriptions of individual functions.

38 39

MPI\_PUBLISH\_NAME(service\_name, info, port\_name)

IN	service_name	a service name to associate with the port (string)
IN	info	implementation-specific information (handle)
IN	port_name	a port name (string)

```
<sup>46</sup> C binding
```

```
<sup>47</sup> int MPI_Publish_name(const char *service_name, MPI_Info info,
<sup>48</sup> const char *port_name)
```

Fortran 2008 binding	
MPI_Publish_name(service_name, info, port_name, ierror)	:
CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name	:
TYPE(MPI_Info), INTENT(IN) :: info	
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	
Fortuge binding	
Fortran binding	
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)	:
CHARACTER*(*) SERVICE_NAME, PORT_NAME	
INTEGER INFO, IERROR	1

This routine publishes the pair (port\_name, service\_name) so that an application may retrieve a system-supplied port\_name using a well-known service\_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port\_name, service\_name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the info argument to MPI\_PUBLISH\_NAME.

MPI permits publishing more than one service\_name for a single port\_name. On the other hand, if service\_name has already been published within the scope determined by info, the behavior of MPI\_PUBLISH\_NAME is undefined. An MPI implementation may, through a mechanism in the info argument to MPI\_PUBLISH\_NAME, provide a way to allow multiple servers with the same service in the same scope. In this case, an implementation-defined policy will determine which of several port names is returned by MPI\_LOOKUP\_NAME.

Note that while service\_name has a limited scope, determined by the implementation, port\_name always has global scope within the communication universe used by the implementation (i.e., it is globally unique).

port\_name should be the name of a port established by MPI\_OPEN\_PORT and not yet released by MPI\_CLOSE\_PORT. If it is not, the result is undefined.

Advice to implementors. In some cases, an MPI implementation may use a name service that a user can also access directly. In this case, a name published by MPI could easily conflict with a name published by a user. In order to avoid such conflicts, MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

The following situation is problematic but unavoidable, if we want to allow implementations to use nameservers. Suppose there are multiple instances of "ocean" running on a machine. If the scope of a service name is confined to a job, then multiple oceans can coexist. If an implementation provides site-wide scope, however, multiple instances are not possible as all calls to MPI\_PUBLISH\_NAME after the first may fail. There is no universal solution to this.

To handle these situations, a high-quality implementation should make it possible to limit the domain over which names are published. (*End of advice to implementors.*)

 $^{24}$ 

 $^{31}$ 

#### 540 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1 MPI\_UNPUBLISH\_NAME(service\_name, info, port\_name) 2 IN service\_name a service name (string) 3 IN info implementation-specific information (handle) 4  $\mathbf{5}$ IN port\_name a port name (string) 6 7C binding 8 int MPI\_Unpublish\_name(const char \*service\_name, MPI\_Info info, 9 const char \*port\_name) 10 Fortran 2008 binding 11 MPI\_Unpublish\_name(service\_name, info, port\_name, ierror) 12CHARACTER(LEN=\*), INTENT(IN) :: service\_name, port\_name 13 TYPE(MPI\_Info), INTENT(IN) :: info 14INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1516Fortran binding 17MPI\_UNPUBLISH\_NAME(SERVICE\_NAME, INFO, PORT\_NAME, IERROR) 18 CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME 19INTEGER INFO, IERROR 20This routine unpublishes a service name that has been previously published. Attempt-21ing to unpublish a name that has not been published or has already been unpublished is 22 erroneous and is indicated by the error class MPI\_ERR\_SERVICE. 23All published names must be unpublished before the corresponding port is closed and  $^{24}$ before the publishing process exits. The behavior of MPI\_UNPUBLISH\_NAME is implemen-25tation dependent when a process tries to unpublish a name that it did not publish. 26If the info argument was used with MPI\_PUBLISH\_NAME to tell the implementation 27how to publish names, the implementation may require that info passed to 28MPI\_UNPUBLISH\_NAME contain information to tell the implementation how to unpublish 29a name. 30  $^{31}$ 32 MPI\_LOOKUP\_NAME(service\_name, info, port\_name) 33 34IN service\_name a service name (string) 35 IN info implementation-specific information (handle) 36 OUT port\_name a port name (string) 37 38 C binding 39 int MPI\_Lookup\_name(const char \*service\_name, MPI\_Info info, 40char \*port\_name) 41 42Fortran 2008 binding 43MPI\_Lookup\_name(service\_name, info, port\_name, ierror) 44CHARACTER(LEN=\*), INTENT(IN) :: service\_name 45TYPE(MPI\_Info), INTENT(IN) :: info 46CHARACTER(LEN=MPI\_MAX\_PORT\_NAME), INTENT(OUT) :: port\_name 47 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 48

#### Fortran binding

```
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
```

This function retrieves a port\_name published by MPI\_PUBLISH\_NAME with service\_name. If service\_name has not been published, it raises an error in the error class MPI\_ERR\_NAME. The application must supply a port\_name buffer large enough to hold the largest possible port name (see discussion above under MPI\_OPEN\_PORT).

If an implementation allows multiple entries with the same service\_name within the same scope, a particular port\_name is chosen in a way determined by the implementation.

If the info argument was used with MPI\_PUBLISH\_NAME to tell the implementation how to publish names, a similar info argument may be required for MPI\_LOOKUP\_NAME.

#### 11.9.5 Reserved Key Values

The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

- "ip\_port" Value contains IP port number at which to establish a port. (Reserved for MPI\_OPEN\_PORT only).
- "ip\_address" Value contains IP address at which to establish a port. If the address is not a valid IP address of the host on which the MPI\_OPEN\_PORT call is made, the results are undefined. (Reserved for MPI\_OPEN\_PORT only).

### 11.9.6 Client/Server Examples

Example 11.21 Simplest Example—Completely Portable.

The following example shows the simplest way to use the client/server interface. It does not use service names at all.

On the server side:

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
```

The server prints out the port name to the terminal and the user must type it in when starting up the client (assuming the MPI implementation supports stdin such that this works). On the client side:

```
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
```

 $\mathbf{2}$ 

## 542 CHAPTER 11. PROCESS INITIALIZATION, CREATION, AND MANAGEMENT

1

 $\mathbf{2}$ 

3

4 5

6

7 8

9

10

11 12 13

14

15

16

17 18 19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32 33

34

35

36

37

38

39

40

41

42

43

44

45

46

47 48 **Example 11.22** Ocean/Atmosphere—Relies on Name Publishing In this example, the "ocean" application is the "server" side of a coupled ocean-atmosphere climate model. It assumes that the MPI implementation publishes names. MPI\_Open\_port(MPI\_INFO\_NULL, port\_name); MPI\_Publish\_name("ocean", MPI\_INFO\_NULL, port\_name); MPI\_Comm\_accept(port\_name, MPI\_INFO\_NULL, 0, MPI\_COMM\_SELF, &intercomm); /\* do something with intercomm \*/ MPI\_Unpublish\_name("ocean", MPI\_INFO\_NULL, port\_name); On the client side: MPI\_Lookup\_name("ocean", MPI\_INFO\_NULL, port\_name); MPI\_Comm\_connect(port\_name, MPI\_INFO\_NULL, 0, MPI\_COMM\_SELF, &intercomm); **Example 11.23** Simple Client-Server Example This is a simple example; the server accepts only a single connection at a time and serves that connection until the client requests to be disconnected. The server is a single process. Here is the server. It accepts a single connection and then processes data until it receives a message with tag 1. A message with tag 0 tells the server to exit. #include "mpi.h" int main(int argc, char \*argv[]) { MPI\_Comm client; MPI\_Status status; char port\_name[MPI\_MAX\_PORT\_NAME]; double buf[MAX\_DATA]; int size, again; MPI\_Init(&argc, &argv); MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); if (size != 1) error(FATAL, "Server too big"); MPI\_Open\_port(MPI\_INFO\_NULL, port\_name); printf("server available at %s\n", port\_name); while (1) { MPI\_Comm\_accept(port\_name, MPI\_INFO\_NULL, 0, MPI\_COMM\_WORLD, &client); again = 1;while (again) { MPI\_Recv(buf, MAX\_DATA, MPI\_DOUBLE, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, client, &status); switch (status.MPI\_TAG) { case 0: MPI\_Comm\_free(&client);

```
MPI_Close_port(port_name);
                        MPI_Finalize();
                         return 0;
                case 1: MPI_Comm_disconnect(&client);
                         again = 0;
                         break;
                case 2: /* do something */
                 . . .
                default:
                         /* Unexpected message type */
                        MPI_Abort(MPI_COMM_WORLD, 1);
                }
            }
        }
}
Here is the client.
#include "mpi.h"
int main(int argc, char *argv[])
{
    MPI_Comm server;
    int done = 0;
    double buf[MAX_DATA];
    char port_name[MPI_MAX_PORT_NAME];
    MPI_Init(&argc, &argv);
    strcpy(port_name, argv[1]);/* assume server's name is cmd-line arg */
    MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                     &server);
    while (!done) {
        tag = 2; /* Action to perform */
        MPI_Send(buf, n, MPI_DOUBLE, 0, tag, server);
        /* etc */
        }
    MPI_Send(buf, 0, MPI_DOUBLE, 0, 1, server);
    MPI_Comm_disconnect(&server);
    MPI_Finalize();
    return 0;
}
```

1

 $\mathbf{2}$ 

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17 18

19

20

21

22

23

 $^{24}$ 

25 26

27

28 29

30

31 32

33

34

35

36

37

38

39

40

 $^{41}$ 

# 11.10 Other Functionality

# 11.10.1 Universe Size

1 2

3 4

5

6

7

8

9

20

21 22

23

24 25

26

27

Many "dynamic" MPI applications are expected to exist in a static runtime environment, in which resources have been allocated before the application is run. When running one of these quasi-static applications, the user (or possibly a batch system) will usually specify a number of processes to start and a total number of processes that are expected. An application simply needs to know how many slots there are, i.e., how many processes it should spawn.

10 MPI provides an attribute on MPI\_COMM\_WORLD, MPI\_UNIVERSE\_SIZE, that allows the 11 application to obtain this information in a portable manner. This attribute indicates the 12total number of processes that are expected. In Fortran, the attribute is the integer value. 13 In C, the attribute is a pointer to the integer value. An application typically subtracts 14the size of MPI\_COMM\_WORLD from MPI\_UNIVERSE\_SIZE to find out how many processes it 15should spawn. MPI\_UNIVERSE\_SIZE is initialized in MPI\_INIT and is not changed by MPI. If 16defined, it has the same value on all processes of MPI\_COMM\_WORLD. MPI\_UNIVERSE\_SIZE 17is determined by the application startup mechanism in a way not specified by MPI. (The 18 size of MPI\_COMM\_WORLD is another example of such a parameter.) 19

Possibilities for how MPI\_UNIVERSE\_SIZE might be set include:

- A -universe\_size argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to MPI\_COMM\_SPAWN through the info argument.

An implementation must document how MPI\_UNIVERSE\_SIZE is set. An implementation may not support the ability to set MPI\_UNIVERSE\_SIZE, in which case the attribute MPI\_UNIVERSE\_SIZE is not set.

<sup>31</sup> MPI\_UNIVERSE\_SIZE is a recommendation, not necessarily a hard limit. For instance, <sup>32</sup> some implementations may allow an application to spawn 50 processes per processor, if <sup>34</sup> they are requested. However, it is likely that the user only wants to spawn one process per <sup>35</sup> processor.

<sup>35</sup> MPI\_UNIVERSE\_SIZE is assumed to have been specified when an application was started, <sup>36</sup> and is in essence a portable mechanism to allow the user to pass to the application (through <sup>37</sup> the MPI process startup mechanism, such as mpiexec) a piece of critical runtime informa-<sup>38</sup> tion. Note that no interaction with the runtime environment is required. If the runtime <sup>40</sup> environment changes size while an application is running, MPI\_UNIVERSE\_SIZE is not up-<sup>41</sup> dated, and the application must find out about the change through direct communication <sup>42</sup> with the runtime system.

42 43

44

# 11.10.2 Singleton MPI Initialization

<sup>45</sup> A high-quality implementation will allow any process (including those not started with a <sup>46</sup> "parallel application" mechanism) to become an MPI process by calling MPI\_INIT,

<sup>47</sup> MPI\_INIT\_THREAD, or MPI\_SESSION\_INIT. Such a process can then connect to other MPI
 <sup>48</sup> processes using the MPI\_COMM\_ACCEPT and MPI\_COMM\_CONNECT routines, or spawn

other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

Advice to implementors. Special coordination is required to start MPI processes belonging to the same MPI\_COMM\_WORLD in the case of the World Model, or the same "mpi://WORLD" process set in the Sessions Model. The processes must be started at the "same" time, they must have a mechanism to establish communication, etc. Either the user or the operating system must take special steps beyond simply starting processes.

Considering the World Model, when an application enters MPI\_INIT, clearly it must be able to determine if these special steps were taken. If a process enters MPI\_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI\_COMM\_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI\_COMM\_WORLD has size 1.

In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either

- 1. Create the environment (e.g., start a daemon) or
- 2. Raise an error if it cannot create the environment and the environment has not been started independently.

A high-quality implementation will try to create a singleton MPI process and not raise an error.(*End of advice to implementors.*)

# 11.10.3 MPI\_APPNUM

There is a predefined attribute MPI\_APPNUM of MPI\_COMM\_WORLD. In Fortran, the attribute is an integer value. In C, the attribute is a pointer to an integer value. If a process was spawned with MPI\_COMM\_SPAWN\_MULTIPLE, MPI\_APPNUM is the command number that generated the current process. Numbering starts from zero. If a process was spawned with MPI\_COMM\_SPAWN, it will have MPI\_APPNUM equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementationspecific startup mechanism that can handle multiple process specifications, MPI\_APPNUM should be set to the number of the corresponding process specification. In particular, if it is started with

mpiexec spec0 [: spec1 : spec2 : ...]

MPI\_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI\_COMM\_SPAWN or

MPI\_COMM\_SPAWN\_MULTIPLE, and MPI\_APPNUM does not make sense in the context of the implementation-specific startup mechanism, MPI\_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI\_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

"appnum" Value contains an integer that overrides the default value for MPI\_APPNUM in the child.

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11 12

13

14 15

16

17

18

19

20 21

22

23

 $^{24}$ 

25 26

27 28

29

30

 $^{31}$ 

32

33

34

35

36

37

38 39

40

41

42

43

44

45

46 47

*Rationale.* When a single application is started, it is able to figure out how many processes there are by looking at the size of MPI\_COMM\_WORLD. An application consisting of multiple SPMD sub-applications has no way to find out how many sub-applications there are and to which sub-application the process belongs. While there are ways to figure it out in special cases, there is no general mechanism. MPI\_APPNUM provides such a general mechanism. (End of rationale.)

7 8

1

 $\mathbf{2}$ 

3

4

5

6

9

11

16

17

18 19

20

21

22

23

 $^{24}$ 

2526

27

2829

30

31

32 33

34

35

36 37

38

39 40

41

42

43

4445

#### 11.10.4 **Releasing Connections**

Before a client and a server connect, they are independent MPI applications. An error in 10 one does not affect the other. After establishing a connection with MPI\_COMM\_CONNECT and MPI\_COMM\_ACCEPT, an error in one may affect the other. It is desirable for a client 12and a server to be able to disconnect, so that an error in one will not affect the other. 13 Similarly, it might be desirable for a parent and child to disconnect, so that errors in the 14child do not affect the parent, or vice-versa. 15

- Two processes are **connected** if there is a communication path (direct or indirect) between them. More precisely:
  - 1. Two processes are connected if
    - (a) they both belong to the same communicator (inter- or intra-, including MPI\_COMM\_WORLD) or
    - (b) they have previously belonged to a communicator that was freed with MPI\_COMM\_FREE instead of MPI\_COMM\_DISCONNECT or
    - (c) they both belong to the group of the same window or file handle.
  - 2. If A is connected to B and B to C, then A is connected to C.
  - Two processes are **disconnected** (also **independent**) if they are not connected.
  - By the above definitions, connectivity is a transitive property, and divides the universe of MPI processes into disconnected (independent) sets (equivalence classes) of processes.
  - Processes which are connected, but do not share the same MPI\_COMM\_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI\_COMM\_DISCONNECT.
  - The following additional rules apply to MPI routines in other chapters:
  - MPI\_FINALIZE is collective over a set of connected processes.
  - MPI\_ABORT does not abort independent processes. It may abort all processes in the caller's MPI\_COMM\_WORLD (ignoring its comm argument). Additionally, it may abort connected processes as well, though it makes a "best attempt" to abort only the processes in comm.
  - If a process terminates without calling MPI\_FINALIZE, independent processes are not affected but the effect on connected processes is not defined.
- 4647 48

Advice to implementors. In practice, it may be difficult to distinguish between an MPI process failure and an erroneous program that terminates without calling an MPI finalization function: an implementation that defines semantics for process failure management may have to exhibit the behavior defined for MPI process failures with such erroneous programs. A high quality implementation should exhibit a different behavior for erroneous programs and MPI process failures. (End of advice to *implementors.*)

MPI_COMM_DISCONNECT(comm) INOUT comm communicator (handle) C binding int MPI_Comm_disconnect(MPI_Comm *comm) Fortran 2008 binding MPI_COmm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users</i> .) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. ( <i>End of rationale</i> .)		8
<pre>MPI_COMM_DISCONNECT(comm) NOUT comm communicator (handle) C binding int MPI_Comm_disconnect(MPI_Comm *comm) Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER coMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</pre>		9
<pre>INOUT comm communicator (handle) C binding int MPI_Comm_disconnect(MPI_Comm *comm) Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm, interror) TYPE(MPI_Comm, interror) TYPE(MPI_Comm, interror) integer, OPTIONAL, INTENT(IOUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of nationale.)</pre>	MDL COMM DISCONNECT(comm)	10
C binding int MPI_Comm_disconnect(MPI_Comm *comm) Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INUUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)		11
<pre>C binding int MPI_Comm_disconnect(MPI_Comm *comm) Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</pre>	INOUT comm communicator (handle)	12
<pre>C Dinding int MPI_Comm_disconnect(MPI_Comm *comm) Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</pre>		13
<pre>Fortran 2008 binding MPI_COmm_disconnect(vPI_comm *comm) Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</pre>	C binding	14
<pre>Fortran 2008 binding MPI_Comm_disconnect(comm, ierror) TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</pre>	<pre>int MPI_Comm_disconnect(MPI_Comm *comm)</pre>	15
<pre>MPI_Comm_disconnect(comm, ierror)     TYPE(MPI_Comm), INTENT(INOUT) :: comm     INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR)     INTEGER COMM, IERROR     This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.     It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF.     MPI_COMM_DISCONNECT may be called only if all communication is complete and     matched, so that buffered data can be delivered to its destination. This requirement is the     same as for MPI_FINALIZE.     MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it     waits for pending communication to finish internally and enables the guarantee about the     behavior of disconnected processes.     Advice to users. To disconnect two processes you may need to call     MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all     communication paths between the two processes. Note that it may be necessary     to disconnect several communicators (or to free several windows or files) before two     processes are completely independent. (End of advice to users.)     Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that     function explicitly does not wait for pending communication to complete. (End of     rationale.) </pre>	Fortran 2008 binding	16 17
<pre>TYPE(MPI_Comm), INTENT(INOUT) :: comm INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre> Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)		18
<pre>INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</pre>	•	19
<pre>Fortran binding MPI_COMM_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</pre>		20
<ul> <li>MPI_COMM_DISCONNECT (COMM, IERROR) INTEGER COMM, IERROR</li> <li>This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.</li> <li>It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF.</li> <li>MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.</li> <li>MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</li> </ul>	Deuteren hin die e	21
<ul> <li>INTEGER COMM, IERROR</li> <li>This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.</li> <li>It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.</li> <li>MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</li> </ul>		22
This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)	•	23
<ul> <li>deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.</li> <li>It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF.</li> <li>MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.</li> <li>MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</li> </ul>	INTEGER COMM, IERROR	24
collective operation. It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)		25
It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF. MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE. MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes. Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.) Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)		26
<ul> <li>MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.</li> <li>MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</li> </ul>	*	27
<ul> <li>matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.</li> <li>MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (<i>End of advice to users.</i>)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (<i>End of rationale.</i>)</li> </ul>	·	28
<ul> <li>same as for MPI_FINALIZE.</li> <li>MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</li> </ul>		29
<ul> <li>MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</li> </ul>		30
<ul> <li>waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (End of advice to users.)</li> <li>Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)</li> </ul>		31
<ul> <li>behavior of disconnected processes.</li> <li>Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (<i>End of advice to users.</i>)</li> <li><i>Rationale.</i> It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (<i>End of rationale.</i>)</li> </ul>		32
Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users.</i> ) <i>Rationale.</i> It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. ( <i>End of rationale.</i> )		33
Advice to users. To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users.</i> ) <i>Rationale.</i> It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. ( <i>End of rationale.</i> )	behavior of disconnected processes.	34
<ul> <li>MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (<i>End of advice to users.</i>)</li> <li><i>Rationale.</i> It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (<i>End of rationale.</i>)</li> </ul>	Advice to users. To disconnect two processes you may need to call	35 36
communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users.</i> ) <i>Rationale.</i> It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. ( <i>End of rationale.</i> )		30 37
to disconnect several communicators (or to free several windows or files) before two processes are completely independent. ( <i>End of advice to users.</i> ) <i>Rationale.</i> It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. ( <i>End of</i> <i>rationale.</i> )		38
Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)	to disconnect several communicators (or to free several windows or files) before two	39
Rationale. It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (End of rationale.)	processes are completely independent. (End of advice to users.)	40
function explicitly does not wait for pending communication to complete. ( <i>End of rationale.</i> )		41
rationale.)		42
		43
	rationale.)	44
		45
		46
		47
4		48

 $\mathbf{2}$ 

1 11.10.5 Another Way to Establish MPI Communication  $\mathbf{2}$ 3 4 MPI\_COMM\_JOIN(fd, intercomm) 5IN fd socket file descriptor 6 OUT  $\overline{7}$ intercomm new inter-communicator (handle) 8 9 C binding 10 int MPI\_Comm\_join(int fd, MPI\_Comm \*intercomm) 11Fortran 2008 binding 12MPI\_Comm\_join(fd, intercomm, ierror) 13 INTEGER, INTENT(IN) :: fd 14TYPE(MPI\_Comm), INTENT(OUT) :: intercomm 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617Fortran binding 18 MPI\_COMM\_JOIN(FD, INTERCOMM, IERROR) 19INTEGER FD, INTERCOMM, IERROR 20MPI\_COMM\_JOIN is intended for MPI implementations that exist in an environment 21supporting the Berkeley Socket interface [50, 56]. Implementations that exist in an environ-22 ment not supporting Berkeley Sockets should provide the entry point for MPI\_COMM\_JOIN 23and should return MPI\_COMM\_NULL.  $^{24}$ This call creates an inter-communicator from the union of two MPI processes which are 25connected by a socket. MPI\_COMM\_JOIN should normally succeed if the local and remote 26processes have access to the same implementation-defined MPI communication universe. 2728 Advice to users. An MPI implementation may require a specific communication 29 medium for MPI communication, such as a shared memory segment or a special switch. 30 In this case, it may not be possible for two processes to successfully join even if there 31is a socket connecting them and they are using the same MPI implementation. (End 32 of advice to users.) 33 34 Advice to implementors. A high-quality implementation will attempt to establish 35communication over a slow medium if its preferred one is not available. If implemen-36 tations do not do this, they must document why they cannot do MPI communication 37 over the medium used by the socket (especially if the socket is a TCP connection). 38 (End of advice to implementors.) 39 40 fd is a file descriptor representing a socket of type SOCK\_STREAM (a two-way reliable 41 byte-stream connection). Nonblocking I/O and asynchronous notification via SIGIO must 42not be enabled for the socket. The socket must be in a connected state. The socket must 43 be quiescent when MPI\_COMM\_JOIN is called (see below). It is the responsibility of the 44application to create the socket using standard socket API calls. 45MPI\_COMM\_JOIN must be called by the process at each end of the socket. It does not 46 return until both processes have called MPI\_COMM\_JOIN. The two processes are referred 47to as the local and remote processes.

48 UC

MPI uses the socket to bootstrap creation of the inter-communicator, and for nothing else. Upon return from MPI\_COMM\_JOIN, the file descriptor will be open and quiescent (see below).

If MPI is unable to create an inter-communicator, but is able to leave the socket in its original state, with no pending communication, it succeeds and sets intercomm to MPI\_COMM\_NULL.

The socket must be quiescent before MPI\_COMM\_JOIN is called and after MPI\_COMM\_JOIN returns. More specifically, on entry to MPI\_COMM\_JOIN, a read on the socket will not read any data that was written to the socket before the remote process called MPI\_COMM\_JOIN. On exit from MPI\_COMM\_JOIN, a read will not read any data that was written to the socket before the remote process returned from MPI\_COMM\_JOIN. It is the responsibility of the application to ensure the first condition, and the responsibility of the MPI implementation to ensure the second. In a multithreaded application, the application must ensure that one thread does not access the socket while another is calling MPI\_COMM\_JOIN, or call MPI\_COMM\_JOIN concurrently.

Advice to implementors. MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (*End of advice to implementors.*)

MPI\_COMM\_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the result of calling MPI\_COMM\_JOIN on two connected processes (see Section 11.10.4 for the definition of connected) is undefined.

The returned communicator may be used to establish MPI communication with additional processes, through the usual MPI communicator creation mechanisms.  $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

# Chapter 12

# **One-Sided** Communications

## 12.1 Introduction

**Remote Memory Access (RMA)** extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. In such a case, each process can compute what data it needs to access or to update at other processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This distribution may require all processes to participate in a time-consuming global computation, or to poll for potential communication requests to receive and upon which to act periodically. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form A = B(map), where map is a permutation vector, and A, B, and map are distributed in the same manner.

7 8 9

15 16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37 38

39 40

41

42

43 44

 $45 \\ 46$ 

47

48

Message-passing communication achieves two effects: *communication* of data from sender to receiver and *synchronization* of sender with receiver. The RMA design separates these two functions. The following communication calls are provided:

- Remote write: MPI\_PUT, MPI\_RPUT
- Remote read: MPI\_GET, MPI\_RGET
- Remote update: MPI\_ACCUMULATE, MPI\_RACCUMULATE
- Remote read and update: MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP
- Remote atomic swap operations: MPI\_COMPARE\_AND\_SWAP

This chapter refers to an operations set that includes all remote update, remote read and update, and remote atomic swap operations as "accumulate" operations.

1 MPI supports two fundamentally different *memory models*: separate and *unified*. The  $\mathbf{2}$ separate model makes no assumption about memory consistency and is highly portable. 3 This model is similar to that of weakly coherent memory systems: the user must impose 4 correct ordering of memory accesses through synchronization calls. The unified model can  $\mathbf{5}$ exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are 6 commonly available in high-performance systems. The two different models are discussed  $\overline{7}$ in detail in Section 12.4. Both models support several synchronization calls to support 8 different synchronization styles.

<sup>9</sup> The design of the RMA functions allows implementors to take advantage of fast or <sup>10</sup> asynchronous communication mechanisms provided by various platforms, such as coherent <sup>11</sup> or noncoherent shared memory, DMA engines, hardware-supported put/get operations, and <sup>12</sup> communication coprocessors. The most frequently used RMA communication mechanisms <sup>13</sup> can be layered on top of message-passing. However, certain RMA functions might need <sup>14</sup> support for asynchronous communication agents in software (handlers, threads, etc.) in a <sup>15</sup> distributed memory environment.

<sup>16</sup> We shall denote by **origin** the process that performs the call, and by **target** the <sup>17</sup> process in which the memory is accessed. Thus, in a put operation, source = origin and <sup>18</sup> destination = target; in a get operation, source = target and destination = origin.

The use of terms such as nonblocking and local in this chapter follow the usage in
 MPI-3.1, and this chapter has not been updated to follow the definitions in Section 2.4.
 The MPI Forum intends to update this chapter in a subsequent version of the MPI standard to follow the definitions in Section 2.4.

 $23 \\ 24$ 

25

# 12.2 Initialization

<sup>26</sup> MPI provides the following window initialization functions: MPI\_WIN\_CREATE,

<sup>27</sup> MPI\_WIN\_ALLOCATE, MPI\_WIN\_ALLOCATE\_SHARED, and

<sup>20</sup> MPI\_WIN\_CREATE\_DYNAMIC, which are collective on an intra-communicator.

<sup>29</sup> MPI\_WIN\_CREATE allows each process to specify a "window" in its memory that is made accessible to accesses by remote processes. The call returns an opaque object that represents the group of processes that own and access the set of windows, and the attributes of each window, as specified by the initialization call. MPI\_WIN\_ALLOCATE differs from

<sup>34</sup> MPI\_WIN\_CREATE in that the user does not pass allocated memory;

<sup>37</sup> MPI\_WIN\_ALLOCATE returns a pointer to memory allocated by the MPI implementation. <sup>35</sup> MPI\_WIN\_ALLOCATE\_SHARED differs from MPI\_WIN\_ALLOCATE in that the allocated <sup>36</sup> memory can be accessed from all processes in the window's group with direct load/store <sup>37</sup> instructions. Some restrictions may apply to the specified communicator.

<sup>39</sup> MPI\_WIN\_CREATE\_DYNAMIC creates a window that allows the user to dynamically control <sup>39</sup> which memory is exposed by the window.

- 41
- 42

- 44 45
- 46
- 47
- 48

12.2.1	Window Creation		1 2
			3
MPI W	/IN_CREATE(base, size	, disp_unit, info, comm, win)	4
IN	base	initial address of window (choice)	5
			6
IN	size	size of window in bytes (non-negative integer)	7
IN	disp_unit	local unit size for displacements, in bytes (positive integer)	8 9
IN	info	info argument (handle)	10 11
IN	comm	intra-communicator (handle)	12
OUT	win	window object (handle)	13
			14
C bin	ding		15
	0	<pre>*base, MPI_Aint size, int disp_unit, MPI_Info info,</pre>	16
	MPI_Comm cor	mm, MPI_Win *win)	17 18
int MF	PI Win create c(void	l *base, MPI_Aint size, MPI_Aint disp_unit,	19
		fo, MPI_Comm comm, MPI_Win *win)	20
Fontre	m 2009 hinding		21
	n 2008 binding	e, disp_unit, info, comm, win, ierror)	22
		), ASYNCHRONOUS :: base	23
		RESS_KIND), INTENT(IN) :: size	24
	TEGER, INTENT(IN) :		25 26
ΤY	PE(MPI_Info), INTEN	NT(IN) :: info	20
	PE(MPI_Comm), INTEN		28
	PE(MPI_Win), INTENT		29
II	TEGER, OPTIONAL, IN	NTENT(OUT) :: ierror	30
MPI_Wi	n_create(base, size	e, disp_unit, info, comm, win, ierror) !(_c)	31
ΤY	<pre>PE(*), DIMENSION(</pre>	), ASYNCHRONOUS :: base	32
		RESS_KIND), INTENT(IN) :: size, disp_unit	33
	PE(MPI_Info), INTEN		34
	PE(MPI_Comm), INTEN		35
	PE(MPI_Win), INTENT		36 37
Τr	HEGER, UPIIUNAL, IN	NTENT(OUT) :: ierror	38
	n binding		39
		E, DISP_UNIT, INFO, COMM, WIN, IERROR)	40
	<pre>Sype&gt; BASE(*) Strategy (kind-mot_appr)</pre>		41
	TEGER(KIND=MPI_ADDR	RESS_KIND) SIZE NFO, COMM, WIN, IERROR	42
Τľ	TEGER DISF_UNII, IN	WEO, CORRI, WIN, IEMMUR	43

44This is a collective call executed by all processes in the group of comm. It returns 45a window object that can be used by these processes to perform RMA operations. Each 46process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address 4748base. In C, base is the starting address of a memory region. In Fortran, one can pass the

1	first element of a memory region or a whole array, which must be 'simply contiguous' (for
2 3	'simply contiguous,' see also Section 19.1.12). A process may elect to expose no memory
4	by specifying size $= 0$ . The displacement unit argument is provided to facilitate address with motion in <b>PMA</b>
5	The displacement unit argument is provided to facilitate address arithmetic in RMA operations: the target displacement argument of an RMA operation is scaled by the factor
6	
7	disp_unit specified by the target process, at window creation.
8	Rationale. The window size is specified using an address-sized integer, rather than a
9	basic integer type, to allow windows that span more memory than can be described
10	with a basic integer type. ( <i>End of rationale.</i> )
11	
12	Advice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax)
13	sizeof(type), for a window that consists of an array of elements of type type. The
14	latter choice will allow one to use array indices in RMA calls, and have those scaled
15	correctly to byte displacements, even in a heterogeneous environment. (End of advice
16	to users.)
17	
18	The info argument provides optimization hints to the runtime about the expected usage
19	pattern of the window. The following info keys are predefined:
20	"no_locks"—if set to true, then the implementation may assume that passive target synchro-
21	nization (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the given
22	window. This implies that this window is not used for 3-party communication, and
23 24	RMA can be implemented with no (less) asynchronous agent activity at this process.
25	"accumulate_ordering"—controls the ordering of accumulate operations at the target. See
26	Section 12.7.2 for details.
27	"accumulate_ops"—if set to "same_op", the implementation will assume that all concurrent
28 29	accumulate calls to the same target address will use the same operation. If set to
29 30	"same_op_no_op", then the implementation will assume that all concurrent accumulate
31	calls to the same target address will use the same operation or MPI_NO_OP. This can
32	eliminate the need to protect access for certain operation types where the hardware
33	can guarantee atomicity. The default is "same_op_no_op".
34	can guarance atomicity. The default is same_op_no_op .
35	"same_size"—if set to true, then the implementation may assume that the argument size is
36	identical on all processes, and that all processes have provided this info key with the
37	same value.
38	"same_disp_unit"—if set to true, then the implementation may assume that the argument
39	disp_unit is identical on all processes, and that all processes have provided this info
40	key with the same value.
41	key with the same value.
42	Advice to users. The info query mechanism described in Section 12.2.7 can be used
43	to query the specified info arguments for windows that have been passed to a library.
44	It is recommended that libraries check attached info keys for each passed window.
45	(End of advice to users.)
46	
47	The various processes in the group of comm may specify completely different target
48	windows, in location, size, displacement units, and info arguments. As long as all the get,

554

CHAPTER 12. ONE-SIDED COMMUNICATIONS

put and accumulate accesses to a particular process fit their specific target window this should pose no problem. The same area in memory may appear in multiple windows, each associated with a different window object. However, concurrent communications to distinct, overlapping windows may lead to undefined results.

*Rationale.* The reason for specifying the memory that may be accessed from another process in an RMA operation is to permit the programmer to specify what memory can be a target of RMA operations and for the implementation to enforce that specification. For example, with this definition, a server process can safely allow a client process to use RMA operations, knowing that (under the assumption that the MPI implementation does enforce the specified limits on the exposed memory) an error in the client cannot affect any memory other than what was explicitly exposed. (*End of rationale.*)

Advice to users. A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI\_ALLOC\_MEM (Section 9.2) will be better. Also, on some systems, performance is improved when window boundaries are aligned at "natural" boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

Advice to implementors. In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI\_WIN\_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI\_ALLOC\_MEM, or by other, implementation-specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI\_WIN\_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

Implementors should document any performance impact of window alignment. (*End of advice to implementors.*)

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13 14

15

16

17

18

19 20

21

22

23

24

25

26

27

28

29

30 31

32 33

34

35

```
1
     12.2.2 Window That Allocates Memory
\mathbf{2}
3
4
     MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)
5
       IN
                size
                                            size of window in bytes (non-negative integer)
6
       IN
                disp_unit
7
                                            local unit size for displacements, in bytes (positive
8
                                            integer)
9
       IN
                info
                                            info argument (handle)
10
                                            intra-communicator (handle)
       IN
                comm
11
       OUT
                baseptr
                                            initial address of window (choice)
12
13
       OUT
                win
                                            window object returned by call (handle)
14
15
     C binding
16
     int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
17
                    MPI_Comm comm, void *baseptr, MPI_Win *win)
18
19
     int MPI_Win_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,
                    MPI_Comm comm, void *baseptr, MPI_Win *win)
20
21
     Fortran 2008 binding
22
     MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
23
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
25
         INTEGER, INTENT(IN) :: disp_unit
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(C_PTR), INTENT(OUT) :: baseptr
29
         TYPE(MPI_Win), INTENT(OUT) :: win
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) !(_c)
33
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
34
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
         TYPE(MPI_Info), INTENT(IN) :: info
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
37
         TYPE(C_PTR), INTENT(OUT) :: baseptr
38
         TYPE(MPI_Win), INTENT(OUT) :: win
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
41
     MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
42
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
43
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
44
45
         This is a collective call executed by all processes in the group of comm. On each
46
     process, it allocates memory of at least size bytes, returns a pointer to it, and returns a
47
```

window object that can be used by all processes in comm to perform RMA operations. The 48 returned memory consists of size bytes local to each process, starting at address baseptr and is associated with the window as if the user called MPI\_WIN\_CREATE on existing memory. The size argument may be different at each process and size = 0 is valid; however, a library might allocate and expose more memory in order to create a fast, globally symmetric allocation. The discussion of and rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 9.2 also apply to MPI\_WIN\_ALLOCATE; in particular, see the rationale in Section 9.2 for an explanation of the type used for baseptr.

If the Fortran compiler provides TYPE(C\_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) BASEPTR, but with a different specific procedure name:

```
INTERFACE MPI_WIN_ALLOCATE
    SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
            WIN, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
            WIN, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI\_WIN\_ALLOCATE\_CPTR. The implied specific procedure names are described in Section 19.1.5.

*Rationale.* By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (*End of rationale.*)

The info argument can be used to specify hints similar to the info argument for MPI\_WIN\_CREATE and MPI\_ALLOC\_MEM.

The default memory alignment requirements and the "mpi\_minimum\_memory\_alignment" info key described for MPI\_ALLOC\_MEM in Section 9.2 apply to all processes with non-zero size argument.

1 2

3

4

5

6 7

8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27 28

29

30 31

32

33

34

35

36

37 38

39

40

41

42

```
558
                                        CHAPTER 12. ONE-SIDED COMMUNICATIONS
1
     12.2.3 Window That Allocates Shared Memory
\mathbf{2}
3
4
     MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win)
5
       IN
                size
                                            size of local window in bytes (non-negative integer)
6
       IN
                disp_unit
7
                                           local unit size for displacements, in bytes (positive
8
                                           integer)
9
       IN
                info
                                           info argument (handle)
10
                                           intra-communicator (handle)
       IN
                comm
11
       OUT
                baseptr
                                           address of local allocated window segment (choice)
12
13
       OUT
                win
                                            window object returned by the call (handle)
14
15
     C binding
16
     int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
17
                    MPI_Comm comm, void *baseptr, MPI_Win *win)
18
19
     int MPI_Win_allocate_shared_c(MPI_Aint size, MPI_Aint disp_unit,
                    MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)
20
21
     Fortran 2008 binding
22
     MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
23
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
24
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
25
         INTEGER, INTENT(IN) :: disp_unit
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(C_PTR), INTENT(OUT) :: baseptr
29
         TYPE(MPI_Win), INTENT(OUT) :: win
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
33
                    !(_c)
34
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
35
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
         TYPE(MPI_Info), INTENT(IN) :: info
36
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         TYPE(C_PTR), INTENT(OUT) :: baseptr
         TYPE(MPI_Win), INTENT(OUT) :: win
39
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     Fortran binding
42
     MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
43
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
44
         INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
45
46
         This is a collective call executed by all processes in the group of comm. On each
47
     process, it allocates memory of at least size bytes that is shared among all processes in
```

comm, and returns a pointer to the locally allocated segment in **baseptr** that can be used

1 for load/store accesses on the calling process. The locally allocated memory can be the  $\mathbf{2}$ target of load/store accesses by remote processes; the base pointers for other processes 3 can be queried using the function MPI\_WIN\_SHARED\_QUERY. The call also returns a window object that can be used by all processes in comm to perform RMA operations. 4 The size argument may be different at each process and size = 0 is valid. It is the user's 5responsibility to ensure that the communicator comm represents a group of processes that 6  $\overline{7}$ can create a shared memory segment that can be accessed by all processes in the group. The discussions of rationales for MPI\_ALLOC\_MEM and MPI\_FREE\_MEM in Section 9.2 8 9 also apply to MPI\_WIN\_ALLOCATE\_SHARED; in particular, see the rationale in Section 9.2 for an explanation of the type used for **baseptr**. The allocated memory is contiguous across 10 11 process ranks unless the info key "alloc\_shared\_noncontig" is specified. Contiguous across process ranks means that the first address in the memory segment of process i is consecutive 12with the last address in the memory segment of process i-1. This may enable the user to 13 calculate remote address offsets with local information only. 14

If the Fortran compiler provides TYPE(C\_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) BASEPTR, but with a different specific procedure name:

```
INTERFACE MPI_WIN_ALLOCATE_SHARED
   SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
            BASEPTR, WIN, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
   END SUBROUTINE
   SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
            BASEPTR, WIN, IERROR)
       USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
       TYPE(C_PTR) :: BASEPTR
   END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI\_WIN\_ALLOCATE\_SHARED\_CPTR. The implied specific procedure names are described in Section 19.1.5.

The info argument can be used to specify hints similar to the info argument for MPI\_WIN\_CREATE, MPI\_WIN\_ALLOCATE, and MPI\_ALLOC\_MEM. The additional info key "alloc\_shared\_noncontig" allows the library to optimize the layout of the shared memory segments in memory.

Advice to users. If the info key "alloc\_shared\_noncontig" is not set to true, the allocation 44 strategy is to allocate contiguous memory across process ranks. This may limit the 45 performance on some architectures because it does not allow the implementation to 46 modify the data layout (e.g., padding to reduce access latency). (End of advice to 47 users.) 48

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33

34

35 36

37

38

39

40

41

42

Advice to implementors. If the user sets the info key "alloc\_shared\_noncontig" to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (End of advice to implementors.)

For contiguous shared memory allocations, the default alignment requirements outlined for MPI\_ALLOC\_MEM in Section 9.2 and the "mpi\_minimum\_memory\_alignment" info key apply to the start of the contiguous memory that is returned in baseptr to the first process with non-zero size argument. For noncontiguous memory allocations, the default alignment requirements and the "mpi\_minimum\_memory\_alignment" info key apply to all processes with non-zero size argument.

Advice to users. If the info key "alloc\_shared\_noncontig" is not set to true (or ignored by the MPI implementation), the alignment of the memory returned in baseptr to all but the first process with non-zero size argument depends on the value of the size argument provided by other processes. It is thus the user's responsibility to control the alignment of contiguous memory allocated for these processes by ensuring that each process provides a size argument that is an integral multiple of the alignment required for the application. (End of advice to users.)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified memory model* (see Section 12.4) by utilizing the window synchronization functions (see Section 12.5) or explicitly completing outstanding store accesses (e.g., by calling MPI\_WIN\_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

28

29 30 MPI\_WIN\_SHARED\_QUERY(win, rank, size, disp\_unit, baseptr)

31	IN	win	shared memory window object (handle)
32 33	IN	rank	rank in the group of window win or MPI_PROC_NULL (non-negative integer)
34	OUT	size	size of the window segment (non-negative integer)
35 36 37	OUT	disp_unit	local unit size for displacements, in bytes (positive integer)
38 39 40	OUT	baseptr	address for load/store access to window segment (choice)
40 41	C binding	5	
42	int MPI_W	in_shared_query(MPI_Win w	vin, int rank, MPI_Aint *size,
43		int *disp_unit, void	*baseptr)
44 45 46	int MPI_W	<pre>in_shared_query_c(MPI_Win     MPI_Aint *disp_unit,</pre>	win, int rank, MPI_Aint *size, void *baseptr)
47	Fortran 2	008 binding	
48	MPI_Win_s	hared_query(win, rank, si	ze, disp_unit, baseptr, ierror)

1

2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12 13

14

15

16

17

18

19

USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	1
TYPE(MPI_Win), INTENT(IN) :: win	2
INTEGER, INTENT(IN) :: rank	3
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size	4
INTEGER, INTENT(OUT) :: disp_unit	5
TYPE(C_PTR), INTENT(OUT) :: baseptr	6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
	8
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) !(_c)	9
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	10
TYPE(MPI_Win), INTENT(IN) :: win	11
INTEGER, INTENT(IN) :: rank	12
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size, disp_unit	13
TYPE(C_PTR), INTENT(OUT) :: baseptr	14
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	15
Fortran binding	16
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)	17
INTEGER WIN, RANK, DISP_UNIT, IERROR	18
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR	19
	20
This function queries the process-local address for remote memory segments created	21
with MPI_WIN_ALLOCATE_SHARED. This function can return different process-local ad-	22

with MPI\_WIN\_ALLOCATE\_SHARED. This function can return different process-local addresses for the same physical memory on different processes. The returned memory can be used for load/store accesses subject to the constraints defined in Section 12.7. This function can only be called with windows of flavor MPI\_WIN\_FLAVOR\_SHARED. If the passed window is not of flavor MPI\_WIN\_FLAVOR\_SHARED, the error MPI\_ERR\_RMA\_FLAVOR is raised. When rank is MPI\_PROC\_NULL, the pointer, disp\_unit, and size returned are the pointer, disp\_unit, and size of the memory segment belonging the lowest rank that specified size > 0. If all processes in the group attached to the window specified size = 0, then the call returns size = 0 and a baseptr as if MPI\_ALLOC\_MEM was called with size = 0.

If the Fortran compiler provides TYPE(C\_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI\_ADDRESS\_KIND) BASEPTR, but with a different specific procedure name:

INTERFACE MPI_WIN_SHARED_QUERY	
	36
SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &	37
BASEPTR, IERROR)	38
IMPORT :: MPI_ADDRESS_KIND	39
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	40
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR	40
INTEGER (KIND-MFI_ADDRESS_KIND) SIZE, DASEFIR	41
END SUBROUTINE	42
SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &	42
	43
BASEPTR, IERROR)	44
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	44
	45
IMPORT :: MPI_ADDRESS_KIND	46
INTEGER :: WIN, RANK, DISP_UNIT, IERROR	40
, , _ ,	47
INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE	48

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

1 2 3		IYPE(C_PTR) :: SUBROUTINE RFACE	BASEPTR
4 5 6 7 8		_SHARED_QUER`	me of this overloaded function is $\Upsilon_CPTR$ . The implied specific procedure names are described in
9	12.2.4 V	Vindow of Dynam	ically Attached Memory
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	target of the progra greater sa sided acce linked list allocated. <b>new</b> respect amount of window's predefined routine M memory w	RMA calls at the ammer (only this fety) and the MI ss to such memor using RMA oper In a C or C++ ctively. In MPI-2 f memory and the memory. In addi amount of mem PI_WIN_CREATE without remote sy	quires the user to identify the local memory that may be a e time the window is created. This has advantages for both memory can be updated by one-sided operations and provides PI implementation (special steps may be taken to make one- y more efficient). However, consider implementing a modifiable rations; as new items are added to the list, memory must be program, this memory is typically allocated using malloc or RMA, the programmer must create a window with a predefined en implement routines for allocating memory from within the tion, there is no easy way to handle the situation where the ory turns out to be inadequate. To support this model, the _DYNAMIC creates a window that makes it possible to expose nchronization. It must be used in combination with the local H and MPI_WIN_DETACH.
25 26	MPI_WIN	_CREATE_DYNAI	MIC(info, comm, win)
27 28	IN	info	info argument (handle)
29	IN	comm	intra-communicator (handle)
30 31	OUT	win	window object returned by the call (handle)
32 33 34	C bindin int MPI_N	•	mic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
35 36 37 38 39 40	MPI_Win_o TYPE TYPE TYPE	(MPI_Info), INT (MPI_Comm), INT (MPI_Win), INTE	info, comm, win, ierror) ENT(IN) :: info ENT(IN) :: comm NT(OUT) :: win INTENT(OUT) :: ierror
41 42 43 44		-	INFO, COMM, WIN, IERROR) WIN, IERROR
45 46 47 48	a window	win without mer	l executed by all processes in the group of comm. It returns nory attached. Existing process memory can be attached as ne returns a window object that can be used by these processes to

perform RMA operations on attached memory. Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI\_WIN\_CREATE.

In the case of a window created with MPI\_WIN\_CREATE\_DYNAMIC, the target\_disp for all RMA functions is the address at the target; i.e., the effective window\_base is MPI\_BOTTOM and the disp\_unit is one. For dynamic windows, the target\_disp argument to RMA communication operations is not restricted to non-negative values. Users should use MPI\_GET\_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI\_Aint and result in unexpected values on some platforms. The MPI\_AINT\_ADD and MPI\_AINT\_DIFF functions can be used to safely perform address arithmetic with MPI\_Aint displacements. (*End of advice to users.*)

Advice to implementors. In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type MPI\_AINT (see Table 3.3) is able to store addresses from any process. (*End of advice to implementors.*)

Memory at the target cannot be accessed with this window until that memory has been attached using the function MPI\_WIN\_ATTACH. That is, in addition to using MPI\_WIN\_CREATE\_DYNAMIC to create an MPI window, the user must use MPI\_WIN\_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI\_WIN\_ATTACH(win, base, size)

IN	win	window object (handle)
IN	base	initial address of memory to be attached (choice)
IN	size	size of memory to be attached in bytes (non-negative
		integer)

#### C binding

int MPI\_Win\_attach(MPI\_Win win, void \*base, MPI\_Aint size)

#### Fortran 2008 binding

MPI\_Win\_attach(win, base, size, ierror)
 TYPE(MPI\_Win), INTENT(IN) :: win
 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: base
 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: size
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_WIN\_ATTACH(WIN, BASE, SIZE, IERROR)

 $\mathbf{2}$ 

 $\mathbf{5}$ 

1	INTEGER WIN, IEF	RUB	
2	<pre><pre><pre><pre><pre><pre><pre><pre></pre></pre></pre></pre></pre></pre></pre></pre>		
3	• -	I_ADDRESS_KIND) SIZE	
4	Attaches a local m	emory region beginning at <b>base</b> for remote access within the given	
5		region specified must not contain any part that is already attached	
6 7		at is, attaching overlapping memory concurrently within the same	
8		The argument win must be a window that was created with	
9		NAMIC. The local memory region attached to the window consists	
10		address base. In C, base is the starting address of a memory region.	
11	,	ass the first element of a memory region or a whole array, which	
12		uous' (for 'simply contiguous,' see Section 19.1.12). Multiple (but ory regions may be attached to the same window.	
13	non-overtapping) memo	ry regions may be attached to the same window.	
14	Rationale. Req	uiring that memory be explicitly attached before it is exposed to	
15 16		y other processes can simplify implementations and improve perfor-	
17		y to make memory available for RMA operations without requiring a	
18		IN_CREATE call is needed for some one-sided programming models.	
19	(End of rationale.	)	
20	Advice to users.	Attaching memory to a window may require the use of scarce	
21		ttaching large regions of memory is not recommended in portable	
22		hing memory to a window may fail if sufficient resources are not	
23	,	imilar to the behavior of MPI_ALLOC_MEM.	
24 25		responsible for ensuring that MPI_WIN_ATTACH at the target has	
26		process attempts to target that memory with an MPI RMA call.	
27	0	MA operation to memory that has not been attached to a window	
28	created with MPI.	_WIN_CREATE_DYNAMIC is erroneous. ( <i>End of advice to users.</i> )	
29	Advice to implem	entors. A high-quality implementation will attempt to make as	
30	much memory available for attaching as possible. Any limitations should be docu-		
31 32	mented by the im	plementor. (End of advice to implementors.)	
33	Attaching memory	is a local operation as defined by MPI, which means that the call	
34	0 0	npletes without requiring any MPI routine to be called in any other	
35		be detached with the routine MPI_WIN_DETACH. After memory has	
36	, <b>v</b>	not be the target of an $MPI\xspace$ metal operation on that window (unless	
37	the memory is re-attack	ned with MPI_WIN_ATTACH).	
38			
39 40	MPI_WIN_DETACH(wir	n, base)	
41	IN win	window object (handle)	
42	IN base	initial address of memory to be detached (choice)	
43		mitial address of memory to be detached (Choice)	
44	C binding		
45		<pre>/PI_Win win, const void *base)</pre>	
46 47			
48	Fortran 2008 binding MPI_Win_detach(win,	-	
	In I_WIN_Getach(WIN,	Dabe, ICIIVI/	

TYPE(MPI_Win), INTENT(IN) :: win	1
TYPE(*), DIMENSION(), ASYNCHRONOUS :: base	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3 4
Fortran binding	5
MPI_WIN_DETACH(WIN, BASE, IERROR)	6
INTEGER WIN, IERROR	7
<type> BASE(*)</type>	8
Detaches a previously attached memory region beginning at base. The arguments base	9
and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.	10 11
Advice to users. Detaching memory may permit the implementation to make more	12
efficient use of special memory or provide memory that may be needed by a subsequent	13
MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed.	14
Memory should be detached before it is freed by the user. (End of advice to users.)	15
	16
Memory becomes detached when the associated dynamic memory window is freed, see Section 12.2.5.	17
Section 12.2.5.	18
12.2.5 Window Destruction	19
	20 21
	21
MPI_WIN_FREE(win)	22
	24
INOUT win window object (handle)	25
	26
C binding	27
<pre>int MPI_Win_free(MPI_Win *win)</pre>	28
Fortran 2008 binding	29
MPI_Win_free(win, ierror)	30
TYPE(MPI_Win), INTENT(INOUT) :: win	31
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	32
Fortran binding	33
MPI_WIN_FREE(WIN, IERROR)	34
INTEGER WIN, IERROR	35 36
Errors the mindem chiest win and returns a null handle (equal to MDI WIN NULL)	37
Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This is a collective call executed by all processes in the group associated with win.	38
MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement	39
in RMA communications on window win: e.g., the process has called	40
MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST	41
or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called	42
MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. The memory associated	43
with windows created by a call to MPI_WIN_CREATE may be freed after the call returns. If	44
the window was created with MPI_WIN_ALLOCATE, MPI_WIN_FREE will free the window	45
memory that was allocated in MPI_WIN_ALLOCATE. If the window was created with	46
MPI_WIN_ALLOCATE_SHARED, MPI_WIN_FREE will free the window memory that was	47
allocated in MPI_WIN_ALLOCATE_SHARED.	48

Freeing a window that was created with a call to MPI\_WIN\_CREATE\_DYNAMIC detaches all associated memory; i.e., it has the same effect as if all attached memory was detached by calls to MPI\_WIN\_DETACH.

Advice to implementors. MPI\_WIN\_FREE requires a barrier synchronization: no process can return from free until all processes in the group of win call free. This ensures that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. The only exception to this rule is when the user sets the "no\_locks" info key to "true" when creating the window. In that case, an MPI implementation may free the local window without barrier synchronization. (*End of advice to implementors.*)

11 12

13

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

# 12.2.6 Window Attributes

 $_{\rm 14}$   $\,$   $\,$  The following attributes are cached with a window when the window is created.

15		. 1 1 11
16	MPI_WIN_BASE	window base address.
17	MPI_WIN_SIZE	window size, in bytes.
	MPI_WIN_DISP_UNIT	displacement unit associated with the window.
18	MPI_WIN_CREATE_FLAVOR	how the window was created.
19	MPI_WIN_MODEL	memory model for window.
20		memory model for window.
21	In C, calls to MPI_Win_get_attr(win	, MPI_WIN_BASE, &base, &flag),
22	MPI_Win_get_attr(win, MPI_WIN_SIZE,	&size, &flag),
23	MPI_Win_get_attr(win, MPI_WIN_DISP_	UNIT, &disp_unit, &flag),
24	MPI_Win_get_attr(win, MPI_WIN_CREA	TE_FLAVOR, &create_kind, &flag), and
25	MPI_Win_get_attr(win, MPI_WIN_MOD	EL, &memory_model, &flag) will return in base a
26	pointer to the start of the window win,	and will return in size, disp_unit, create_kind, and
27	memory_model pointers to the size, disp	lacement unit of the window, the kind of routine
28	used to create the window, and the men	nory model, respectively. A detailed listing of the
29	type of the pointer in the attribute value	argument to $MPI_WIN_GET_ATTR$ and
30	MPI_WIN_SET_ATTR is shown in Table	12.1.

Attribute	C Type
MPI_WIN_BASE	void *
MPI_WIN_SIZE	MPI_Aint *
MPI_WIN_DISP_UNIT	int *
MPI_WIN_CREATE_FLAVOR	int *
MPI_WIN_MODEL	int *

<sup>39</sup> Table 12.1: C types of attribute value argument to MPI\_WIN\_GET\_ATTR and

40 MPI\_WIN\_SET\_ATTR

- <sup>42</sup> In Fortran, calls to MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_BASE, base, flag, ierror),
- <sup>43</sup> MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_SIZE, size, flag, ierror),
- <sup>44</sup> MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_DISP\_UNIT, disp\_unit, flag, ierror),
- <sup>45</sup> MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_CREATE\_FLAVOR, create\_kind, flag, ierror), and
- <sup>46</sup> MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_MODEL, memory\_model, flag, ierror) will return in
- $\frac{47}{48}$  base, size, disp\_unit, create\_kind, and memory\_model the (integer representation of) the

base address, the size, the displacement unit of the window win, the kind of routine used to create the window, and the memory model, respectively.

The values of create\_kind are

MPI WIN FLAVOR CREATE	Window was created with MPI_WIN_CREATE.
MPI_WIN_FLAVOR_ALLOCATE	Window was created with MPI_WIN_ALLOCATE.
MPI_WIN_FLAVOR_DYNAMIC	Window was created with
	MPI_WIN_CREATE_DYNAMIC.
MPI_WIN_FLAVOR_SHARED	Window was created with
	MPI_WIN_ALLOCATE_SHARED.

The values of memory\_model are MPI\_WIN\_SEPARATE and MPI\_WIN\_UNIFIED. The meaning of these is described in Section 12.4.

In the case of windows created with MPI\_WIN\_CREATE\_DYNAMIC, the base address is MPI\_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section 7.7.3.) The value returned for an attribute on a window is constant over the lifetime of the window.

The other "window attribute," namely the group of processes attached to the window, can be retrieved using the call below.

MPI\_WIN\_GET\_GROUP(win, group)

IN	win	window object (handle)
OUT	group	group of processes which share access to the window
		(handle)

C binding

int MPI\_Win\_get\_group(MPI\_Win win, MPI\_Group \*group)

Fortran 2008 binding

```
MPI_Win_get_group(win, group, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

MPI\_WIN\_GET\_GROUP(WIN, GROUP, IERROR) INTEGER WIN, GROUP, IERROR

MPI\_WIN\_GET\_GROUP returns a duplicate of the group of the communicator used to create the window associated with win. The group is returned in group.

# 12.2.7 Window Info

Hints specified via info (see Section 10) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance definition or use system resources more efficiently. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and that place definition (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and the mathematical to MPI\_WIN\_GET\_INFO) and the mathematical to MPI\_WIN\_GET\_INFO (i.e., are returned by a call to MPI\_WIN\_GET\_INFO) and the m

 $\mathbf{2}$ 

 $41 \\ 42$ 

1 a restriction on the behavior of the application. Hints are specified on a per window basis,  $\mathbf{2}$ in window creation functions and MPI\_WIN\_SET\_INFO, via the opaque info object. When 3 an info object that specifies a subset of valid hints is passed to MPI\_WIN\_SET\_INFO there 4 will be no effect on previously set or default hints that the info does not specify. 5

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for the hint. (End of advice to implementors.)

```
12
13
14
```

15

26

30

38

41

6

 $\overline{7}$ 

8 9

10 11

MPI\_WIN\_SET\_INFO(win, info)

16	INOUT	win	window object (handle)
17	IN	info	info argument (handle)
18			- , , ,
19	C binding	<pre>c</pre>	
20 21	int MPI_W	, in_set_info(MPI_Win win,	MPI_Info info)

```
Fortran 2008 binding
22
```

```
MPI_Win_set_info(win, info, ierror)
23
         TYPE(MPI_Win), INTENT(IN) :: win
^{24}
```

```
TYPE(MPI_Info), INTENT(IN) :: info
25
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
27
     Fortran binding
28
```

MPI\_WIN\_SET\_INFO(WIN, INFO, IERROR) 29

```
INTEGER WIN, INFO, IERROR
```

 $^{31}$ MPI\_WIN\_SET\_INFO updates the hints of the window associated with win using the 32 hints provided in info. This operation has no effect on previously set or defaulted hints 33 that are not specified by info. It also has no effect on previously set or defaulted hints that 34are specified by info, but are ignored by the MPI implementation in this call to MPI\_WIN\_SET\_INFO. The call is collective on the group of win. The info object may be 35 36 different on each process, but any info entries that an implementation requires to be the 37 same on all processes must appear with the same value in each process's info object.

Some info items that an implementation can use when it creates Advice to users. 39 a window cannot easily be changed once the window has been created. Thus, an 40implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a 42call to MPI\_WIN\_SET\_INFO. MPI\_WIN\_GET\_INFO can be used to determine whether 43 info changes were ignored by the implementation. (End of advice to users.) 44

- 45
- 46
- 47
- 48

MPI\_WIN\_GET\_INFO(win, info\_used)

IN	win	window object (handle)
OUT	info_used	new info object (handle)

#### C binding

int MPI\_Win\_get\_info(MPI\_Win win, MPI\_Info \*info\_used)

#### Fortran 2008 binding

MPI\_Win\_get\_info(win, info\_used, ierror)
 TYPE(MPI\_Win), INTENT(IN) :: win
 TYPE(MPI\_Info), INTENT(OUT) :: info\_used
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_WIN\_GET\_INFO(WIN, INFO\_USED, IERROR) INTEGER WIN, INFO\_USED, IERROR

MPI\_WIN\_GET\_INFO returns a new info object containing the hints of the window associated with win. The current setting of all hints related to this window is returned in info\_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info\_used via MPI\_INFO\_FREE.

# 12.3 Communication Calls

MPI supports the following RMA communication calls: MPI\_PUT and MPI\_RPUT transfer data from the caller memory (origin) to the target memory; MPI\_GET and MPI\_RGET transfer data from the target memory to the caller memory; MPI\_ACCUMULATE and MPI\_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI\_GET\_ACCUMULATE,

MPI\_RGET\_ACCUMULATE, and MPI\_FETCH\_AND\_OP perform atomic read-modify-write and return the data before the accumulate operation; and MPI\_COMPARE\_AND\_SWAP performs a remote atomic compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 12.5. Transfers can also be completed with calls to flush routines; see Section 12.5.4 for details. For the MPI\_RPUT, MPI\_RGET, MPI\_RACCUMULATE, and MPI\_RGET\_ACCUMULATE calls, the transfer can be locally completed by using the MPI test or wait operations described in Section 3.7.3.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call until the operation completes at the origin.

The resulting data values, or outcome, of concurrent conflicting accesses to the same 46 memory locations is undefined; if a location is updated by a put or accumulate operation, 47 then the outcome of loads or other RMA operations is undefined until the updating operation 48

1has completed at the target. There is one exception to this rule; namely, the same location  $\mathbf{2}$ can be updated by several concurrent accumulate calls, the outcome being as if these updates 3 occurred in some order. In addition, the outcome of concurrent load/store and RMA updates 4 to the same memory location is undefined. These restrictions are described in more detail  $\mathbf{5}$ in Section 12.7. 6 The calls use general datatype arguments to specify communication buffers at the origin  $\overline{7}$ and at the target. Thus, a transfer operation may also gather data at the source and scatter 8 it at the destination. However, all arguments specifying both communication buffers are 9 provided by the caller. 10 For all RMA calls, the target process may be identical with the origin process; i.e., a 11process may use an RMA operation to move data in its memory. 12Rationale. The choice of supporting "self-communication" is the same as for message-13 passing. It simplifies some coding, and is very useful with accumulate operations, to 14allow atomic updates of local variables. (End of rationale.) 1516MPI\_PROC\_NULL is a valid target rank in all MPI RMA communication calls. The effect 17is the same as for MPI\_PROC\_NULL in MPI point-to-point communication. After any RMA 18 operation with rank MPI\_PROC\_NULL, it is still necessary to finish the RMA epoch with the 19 synchronization method that started the epoch. 202112.3.1 Put 2223The execution of a put operation is similar to the execution of a send by the origin process 24and a matching receive by the target process. The obvious difference is that all arguments 25are provided by one call—the call executed by the origin process. 262728 MPI\_PUT(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, 29target\_datatype, win) 30 IN origin\_addr initial address of origin buffer (choice)  $^{31}$ IN origin\_count 32 number of entries in origin buffer (non-negative integer) 33 34IN origin\_datatype datatype of each entry in origin buffer (handle) 35 IN target\_rank rank of target (non-negative integer) 36 IN target\_disp displacement from start of window to target buffer 37 (non-negative integer) 38 39 IN target\_count number of entries in target buffer (non-negative 40 integer) 41 IN target\_datatype datatype of each entry in target buffer (handle) 42IN win window object used for communication (handle) 43 4445C binding 46int MPI\_Put(const void \*origin\_addr, int origin\_count, 47MPI\_Datatype origin\_datatype, int target\_rank, 48

1 MPI\_Aint target\_disp, int target\_count,  $\mathbf{2}$ MPI\_Datatype target\_datatype, MPI\_Win win) 3 int MPI\_Put\_c(const void \*origin\_addr, MPI\_Count origin\_count, 4 MPI\_Datatype origin\_datatype, int target\_rank, 5MPI\_Aint target\_disp, MPI\_Count target\_count, 6 MPI\_Datatype target\_datatype, MPI\_Win win) 7 8 Fortran 2008 binding MPI\_Put(origin\_addr, origin\_count, origin\_datatype, target\_rank, 9 10target\_disp, target\_count, target\_datatype, win, ierror) 11 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 1213 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 14INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 15TYPE(MPI\_Win), INTENT(IN) :: win 16INTEGER, OPTIONAL, INTENT(OUT) :: ierror 17MPI\_Put(origin\_addr, origin\_count, origin\_datatype, target\_rank, 18 target\_disp, target\_count, target\_datatype, win, ierror) !(\_c) 19 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 20INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: origin\_count, target\_count 21TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 22 INTEGER, INTENT(IN) :: target\_rank 23INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp  $^{24}$ TYPE(MPI\_Win), INTENT(IN) :: win 25INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2627Fortran binding 28 MPI\_PUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, 29TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, IERROR) 30 <type> ORIGIN\_ADDR(\*) 31INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 32 TARGET\_DATATYPE, WIN, IERROR 33 INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 34 Transfers origin\_count successive entries of the type specified by the origin\_datatype, 35

starting at address origin\_addr on the origin node, to the target node specified by the win, target\_rank pair. The data are written in the target buffer at address target\_addr = window\_base+target\_disp×disp\_unit, where window\_base and disp\_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target\_count and target\_datatype.

The data transfer is the same as that which would occur if the origin process executed <sup>41</sup> a send operation with arguments origin\_addr, origin\_count, origin\_datatype, target\_rank, tag, <sup>42</sup> comm, and the target process executed a receive operation with arguments target\_addr, <sup>43</sup> target\_count, target\_datatype, source, tag, comm, where target\_addr is the target buffer <sup>44</sup> address computed as explained above, the values of tag are arbitrary valid matching tag <sup>45</sup> values, and comm is a communicator for the group of win. <sup>46</sup>

The communication must satisfy the same constraints as for a similar message-passing 47 communication. The target\_datatype may not specify overlapping entries in the target 48

36

37

38

39

<sup>1</sup> buffer. The message sent must fit, without truncation, in the target buffer. Furthermore,
 <sup>2</sup> the target buffer must fit in the target window or in attached memory in a dynamic window.
 <sup>3</sup> The target\_datatype argument is a handle to a datatype object defined at the origin
 <sup>4</sup> process. However, this object is interpreted at the target process: the outcome is as if
 <sup>5</sup> the target datatype object was defined at the target process by the same sequence of calls
 <sup>6</sup> used to define it at the origin process. The target datatype must contain only relative
 <sup>7</sup> displacements, not absolute addresses. The same holds for get and accumulate operations.

8 9

10

11

12

13

21

22

23

 $^{24}$ 

25

26

27

28

Advice to users. The target\_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment if only portable datatypes are used (portable datatypes are defined in Section 2.4).

<sup>14</sup> The performance of a put transfer can be significantly affected, on some systems, by <sup>15</sup> the choice of window location and the shape and location of the origin and target <sup>16</sup> buffer: transfers to a target window in memory allocated by MPI\_ALLOC\_MEM or <sup>17</sup> MPI\_WIN\_ALLOCATE may be much faster on shared memory systems; transfers from <sup>18</sup> contiguous buffers will be faster on most, if not all, systems; the alignment of the <sup>19</sup> communication buffers may also impact performance. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This is important both for debugging purposes and for protection with client-server codes that use RMA. That is, a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an error at the origin call if an out-ofbound situation occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)

29 30

12.3.2	Get		1
			2
MPI_GE	T(origin_addr, origin_count target_datatype, wi	, origin_datatype, target_rank, target_disp, target_count, n)	4 5
OUT	origin_addr	initial address of origin buffer (choice)	6 7
IN	origin_count	number of entries in origin buffer (non-negative integer)	8 9
IN	origin_datatype	datatype of each entry in origin buffer (handle)	10 11
IN	target_rank	rank of target (non-negative integer)	11
IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)	13 14
IN	target_count	number of entries in target buffer (non-negative integer)	15 16
IN	target_datatype	datatype of each entry in target buffer (handle)	17 18
IN	win	window object used for communication (handle)	19
			20
C bindi	0		21 22
int MPI	_Get(void *origin_addr	-	22
		igin_datatype, int target_rank,	23
	•	_disp, int target_count,	24
	MPI_Datatype ta	rget_datatype, MPI_Win win)	23 26
int MPI	_Get_c(void *origin_ad	ldr, MPI_Count origin_count,	27
	MPI_Datatype or:	igin_datatype, int target_rank,	28
		_disp, MPI_Count target_count,	29
	MPI_Datatype ta	rget_datatype, MPI_Win win)	30
Fortran	2008 binding		31
MPI_Get	(origin_addr, origin_c	count, origin_datatype, target_rank,	32
	target_disp, tar	rget_count, target_datatype, win, ierror)	33
TYP	E(*), DIMENSION(), A	SYNCHRONOUS :: origin_addr	34
		igin_count, target_rank, target_count	35
		<pre>IT(IN) :: origin_datatype, target_datatype</pre>	36
		KIND), INTENT(IN) :: target_disp	37 38
	E(MPI_Win), INTENT(IN)		39
TN.I.	EGER, OPTIONAL, INTENT	((UUT) :: ierror	40
MPI_Get	(origin_addr, origin_c	count, origin_datatype, target_rank,	41
	target_disp, tar	<pre>rget_count, target_datatype, win, ierror) !(_c)</pre>	42
TYP	E(*), DIMENSION(), A	SYNCHRONOUS :: origin_addr	43
		ND), INTENT(IN) :: origin_count, target_count	44
		<pre>IT(IN) :: origin_datatype, target_datatype</pre>	45
	EGER, INTENT(IN) :: ta	•	46
		KIND), INTENT(IN) :: target_disp	47
TYP	E(MPI_Win), INTENT(IN)	:: win	48

1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2	Fortran binding
3	MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
4	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
5	<pre><type> ORIGIN_ADDR(*)</type></pre>
6	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
7	TARGET_DATATYPE, WIN, IERROR
8	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
9	
10 11 12 13 14 15	Similar to MPI_PUT, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The origin_datatype may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window or within attached memory in a dynamic window, and the copied data must fit, without truncation, in the origin buffer.
16 17	12.3.3 Examples for Communication Calls
18 19 20	These examples show the use of the MPI_GET function. As all MPI RMA communication functions are nonblocking, they must be completed. In the following, this is accomplished with the routine MPI_WIN_FENCE, introduced in Section 12.5.
21 22 23 24	<b>Example 12.1</b> We show how to implement the generic indirect assignment $A = B(map)$ , where A, B, and map have the same distribution, and map is a permutation. To simplify, we assume a block distribution with equal size blocks.
25 26	SUBROUTINE MAPVALS(A, B, map, m, comm, p) USE MPI
27	INTEGER m, map(m), comm, p
28	REAL A(m), B(m)
29 30	
31	<pre>INTEGER otype(p), oindex(m), &amp; ! used to construct origin datatypes</pre>
32	<pre>ttype(p), tindex(m),</pre>
33	<pre>count(p), total(p), &amp;</pre>
34	disp_int, win, ierr
35	INTEGER(KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
36	
37	! This part does the work that depends on the locations of B.
38	! Can be reused while this does not change
39	
40	CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
41	disp_int = realextent
42	size = m * realextent
43	CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, &
44	comm, win, ierr)
45	I This part does the work that depends on the value of you and
45 46	! This part does the work that depends on the value of map and
	! This part does the work that depends on the value of map and ! the locations of the arrays. ! Can be reused while these do not change

```
! Compute number of entries to be received from each process
DO i=1,p
   count(i) = 0
END DO
DO i=1,m
   j = map(i)/m+1
   count(j) = count(j)+1
END DO
total(1) = 0
DO i=2,p
   total(i) = total(i-1) + count(i-1)
END DO
DO i=1,p
   count(i) = 0
END DO
! compute origin and target indices of entries.
! entry i at current process is received from location
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
! j = 1...p and k = 1...m
DO i=1,m
   j = map(i)/m+1
   k = MOD(map(i), m) + 1
   count(j) = count(j)+1
   oindex(total(j) + count(j)) = i
   tindex(total(j) + count(j)) = k
END DO
! create origin and target datatypes for each get operation
DO i=1,p
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                     oindex(total(i)+1:total(i)+count(i)), &
                                     MPI_REAL, otype(i), ierr)
   CALL MPI_TYPE_COMMIT(otype(i), ierr)
   CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                     tindex(total(i)+1:total(i)+count(i)), &
                                     MPI_REAL, ttype(i), ierr)
   CALL MPI_TYPE_COMMIT(ttype(i), ierr)
END DO
! this part does the assignment itself
CALL MPI_WIN_FENCE(0, win, ierr)
```

1

2 3

4

5

6

7

8

9

10 11

12

13

14

15 16

17

18

19 20

21

22

23

24 25

26

27

28

29

30

31

32 33

34

35

36

37

38

39

40

41

42

43

44 45

46

```
1
     disp_aint = 0
\mathbf{2}
     DO i=1,p
3
         CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 1, ttype(i), win, ierr)
4
     END DO
\mathbf{5}
     CALL MPI_WIN_FENCE(0, win, ierr)
6
7
     CALL MPI_WIN_FREE(win, ierr)
8
     DO i=1,p
9
         CALL MPI_TYPE_FREE(otype(i), ierr)
10
         CALL MPI_TYPE_FREE(ttype(i), ierr)
^{11}
     END DO
12
     RETURN
13
     END
14
15
     Example 12.2 A simpler version can be written that does not require that a datatype be
16
     built for the target buffer. But, one then needs a separate get call for each entry, as illustrated
17
     below. This code is much simpler, but usually much less efficient, for large arrays.
18
19
     SUBROUTINE MAPVALS(A, B, map, m, comm, p)
20
     USE MPI
21
     INTEGER m, map(m), comm, p
22
     REAL A(m), B(m)
23
     INTEGER disp_int, win, ierr
^{24}
     INTEGER(KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
25
26
     CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
27
     disp_int = realextent
28
     size = m * realextent
29
     CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL, &
30
                            comm, win, ierr)
^{31}
32
     CALL MPI_WIN_FENCE(0, win, ierr)
33
     DO i=1,m
34
         j = map(i)/m
35
         disp_aint = MOD(map(i),m)
36
         CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL, win, ierr)
37
     END DO
38
     CALL MPI_WIN_FENCE(0, win, ierr)
39
     CALL MPI_WIN_FREE(win, ierr)
40
     RETURN
41
     END
42
43
     12.3.4 Accumulate Functions
```

45It is often useful in a put operation to combine the data moved to the target process with the 46data that resides at that process, rather than replacing it. This will allow, for example, the 47accumulation of a sum by having all involved processes add their contributions to the sum 48

variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 12.7 for details.

### Accumulate Function

MPI\_ACCUMULATE(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, op, win)

IN	origin_addr	initial address of buffer (choice)	11
IN	origin_count	number of entries in buffer (non-negative integer)	12
IN	origin_datatype	datatype of each entry (handle)	13 14
IN	target_rank	rank of target (non-negative integer)	14
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	16 17
IN	target_count	number of entries in target buffer (non-negative integer)	18 19 20
IN	target_datatype	datatype of each entry in target buffer (handle)	20 21
IN	ор	reduce operation (handle)	22
IN	win	window object (handle)	23 24
			- 24

### C binding

int MPI\_Accumulate(const void \*origin\_addr, int origin\_count, MPI\_Datatype origin\_datatype, int target\_rank, MPI\_Aint target\_disp, int target\_count, MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win) int MPI\_Accumulate\_c(const void \*origin\_addr, MPI\_Count origin\_count, MPI\_Datatype origin\_datatype, int target\_rank, MPI\_Aint target\_disp, MPI\_Count target\_count, MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win) Fortran 2008 binding MPI\_Accumulate(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, op, win, ierror) TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp TYPE(MPI\_Op), INTENT(IN) :: op TYPE(MPI\_Win), INTENT(IN) :: win

INTEGER, OPTIONAL, INTENT(OUT) :: ierror

1	TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr
2	INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
3	TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
4	INTEGER, INTENT(IN) :: target_rank
5	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
6	TYPE(MPI_Op), INTENT(IN) :: op
7	TYPE(MPI_Win), INTENT(IN) :: win
8	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
9	INTEGER, OFFICIARE, INTENT(OOF) TETTOT
	Fortran binding
10	MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
11	
12	TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
13	<type> ORIGIN_ADDR(*)</type>
14	INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
	TARGET_DATATYPE, OP, WIN, IERROR
15	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
16	
17	Accumulate the contents of the origin buffer (as defined by origin_addr, origin_count, and
18	origin_datatype) to the buffer specified by arguments target_count and target_datatype, at
19	offset target_disp, in the target window specified by target_rank and win, using the operation
20	
	op. This is like MPI_PUT except that data is combined into the target area instead of
21	overwriting it.
22	Any of the predefined operations for MPI_REDUCE can be used. User-defined functions
23	cannot be used. For example, if <b>op</b> is MPI_SUM, each element of the origin buffer is added
24	to the corresponding element in the target, replacing the former value in the target.
25	Each datatype argument must be a predefined datatype or a derived datatype, where
26	all basic components are of the same predefined datatype. Both datatype arguments must
27	
	be constructed from the same predefined datatype. The operation <b>op</b> applies to elements of
28	that predefined type. The parameter target_datatype must not specify overlapping entries,
29	and the target buffer must fit in the target window.
30	A new predefined operation, MPI_REPLACE, is defined. It corresponds to the associative
31	function $f(a,b) = b$ ; i.e., the current value in the target memory is replaced by the value
32	supplied by the origin.
33	
	MPI_REPLACE can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE,
34	$MPI\_GET\_ACCUMULATE, MPI\_FETCH\_AND\_OP, \mathrm{and}\ MPI\_RGET\_ACCUMULATE, \mathrm{but}\ \mathrm{not}$
35	in collective reduction operations such as MPI_REDUCE.
36	Advise to seems MDL DUT :=
37	Advice to users. MPI_PUT is a special case of MPI_ACCUMULATE, with the op-
38	eration MPI_REPLACE. Note, however, that MPI_PUT and MPI_ACCUMULATE have
39	different constraints on concurrent updates. (End of advice to users.)
40	<b>Example 12.3</b> We want to compute $B(j) = \sum_{map(i)=j} A(i)$ . The arrays A, B, and map
41	
42	are distributed in the same manner. We write the simple version.
43	SUBROUTINE SUM(A, B, map, m, comm, p)
44	
45	USE MPI
46	INTEGER m, map(m), comm, p, win, ierr, disp_int
47	REAL A(m), B(m)
	INTEGER(KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint
48	

```
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
size = m * realextent
disp_int = realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,
                                                        X.
                    comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
   j = map(i)/m
   disp_aint = MOD(map(i),m)
   CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,
                                                                         &
                       MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 12.2, except that a call to get has been replaced by a call to accumulate. (Note that, if map is one-to-one, the code computes  $B = A(map^{-1})$ , which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 12.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

### Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 12.7 for details). The predefined operation MPI\_REPLACE provides fetch-and-set behavior.

 $\mathbf{2}$ 

1MPI\_GET\_ACCUMULATE(origin\_addr, origin\_count, origin\_datatype, result\_addr,  $\mathbf{2}$ result\_count, result\_datatype, target\_rank, target\_disp, target\_count, 3 target\_datatype, op, win) 4 IN origin\_addr initial address of buffer (choice) 5origin\_count IN number of entries in origin buffer (non-negative 6 integer) 7 8 IN origin\_datatype datatype of each entry in origin buffer (handle) 9 OUT result\_addr initial address of result buffer (choice) 10 result\_count IN number of entries in result buffer (non-negative 11 integer) 12IN result\_datatype datatype of each entry in result buffer (handle) 13 14IN target\_rank rank of target (non-negative integer) 15IN target\_disp displacement from start of window to beginning of 16 target buffer (non-negative integer) 17IN target\_count number of entries in target buffer (non-negative 18 integer) 19 20IN target\_datatype datatype of each entry in target buffer (handle) 21IN reduce operation (handle) ор 22 IN win window object (handle) 232425C binding 26int MPI\_Get\_accumulate(const void \*origin\_addr, int origin\_count, 27MPI\_Datatype origin\_datatype, void \*result\_addr, 28int result\_count, MPI\_Datatype result\_datatype, 29 int target\_rank, MPI\_Aint target\_disp, int target\_count, 30 MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win)  $^{31}$ int MPI\_Get\_accumulate\_c(const void \*origin\_addr, MPI\_Count origin\_count, 32 MPI\_Datatype origin\_datatype, void \*result\_addr, 33 MPI\_Count result\_count, MPI\_Datatype result\_datatype, 34 int target\_rank, MPI\_Aint target\_disp, MPI\_Count target\_count, 35 MPI\_Datatype target\_datatype, MPI\_Op op, MPI\_Win win) 36 37 Fortran 2008 binding 38MPI\_Get\_accumulate(origin\_addr, origin\_count, origin\_datatype, result\_addr, 39 result\_count, result\_datatype, target\_rank, target\_disp, 40 target\_count, target\_datatype, op, win, ierror) 41 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 42INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, 43 target\_count 44 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, result\_datatype, 45target\_datatype 46 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: result\_addr 47 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 48 TYPE(MPI\_Op), INTENT(IN) :: op

TYPE(MPI_Win), INTENT(IN) :: win	1
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	2
	3
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_ad	ddr, 4
result_count, result_datatype, target_rank, target_disp,	5
<pre>target_count, target_datatype, op, win, ierror) !(_c)</pre>	6
<pre>TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>	7
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_cou	unt, $_8$
target_count	9
<pre>TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,</pre>	10
target_datatype	11
<pre>TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr</pre>	12
INTEGER, INTENT(IN) :: target_rank	13
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	14
TYPE(MPI_Op), INTENT(IN) :: op	15
TYPE(MPI_Win), INTENT(IN) :: win	16
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
Fortran binding	18
MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_AI	DDR, <sup>19</sup>
RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,	20
TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)	21
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>	22
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYP	E, <sup>23</sup>
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERRO	$\mathbf{DR}$ $^{24}$
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	25
	26

Accumulate origin\_count elements of type origin\_datatype from the origin buffer ( origin\_addr) to the buffer at offset target\_disp, in the target window specified by target\_rank and win, using the operation op and return in the result buffer result\_addr the content of the target buffer before the accumulation, specified by target\_disp, target\_count, and target\_datatype. The data transferred from origin to target must fit, without truncation, in the target buffer. Likewise, the data copied from target to origin must fit, without truncation, in the result buffer.

The origin and result buffers (origin\_addr and result\_addr) must be disjoint. Each datatype argument must be a predefined datatype or a derived datatype where all basic components are of the same predefined datatype. All datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target\_datatype must not specify overlapping entries, and the target buffer must fit in the target window or in attached memory in a dynamic window. The operation is executed atomically for each basic datatype; see Section 12.7 for details.

Any of the predefined operations for MPI\_REDUCE, as well as MPI\_NO\_OP or MPI\_REPLACE can be specified as op. User-defined functions cannot be used. A new predefined operation, MPI\_NO\_OP, is defined. It corresponds to the associative function f(a, b) = a; i.e., the current value in the target memory is returned in the result buffer at the origin and no operation is performed on the target buffer. When MPI\_NO\_OP is specified as the operation, the origin\_addr, origin\_count, and origin\_datatype arguments are ignored. MPI\_NO\_OP can be used only in MPI\_GET\_ACCUMULATE, MPI\_RGET\_ACCUMULATE, 

	582	CH	HAPTER 12. ONE-SIDED COMMUNICATIONS			
1 2 3	and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE, MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others.					
4 5 6 7	Advice to users. MPI_GET is similar to MPI_GET_ACCUMULATE, with the opera- tion MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have different constraints on concurrent updates. ( <i>End of advice to users.</i> )					
8	Fetch and (	Fetch and Op Function				
9 10 11 12 13 14	The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetch- and-increment or fetch-and-add calls that might be supported by special hardware oper- ations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.					
15 16	MPI_FETC	H_AND_OP(origin_addr, resul	t_addr, datatype, target_rank, target_disp, op, win)			
17 18	IN	origin_addr	initial address of buffer (choice)			
19	OUT	result_addr	initial address of result buffer (choice)			
20 21 22	IN	datatype	datatype of the entry in origin, result, and target buffers (handle)			
22	IN	target_rank	rank of target (non-negative integer)			
24 25	IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)			
26 27	IN	ор	reduce operation (handle)			
28	IN	win	window object (handle)			
29 30	C binding	у.				
31 32 33		etch_and_op(const void *c	origin_addr, void *result_addr, e, int target_rank, MPI_Aint target_disp, in)			
34	Fortran 2	008 binding				
35 36	MPI_Fetch	1 0	<pre>ilt_addr, datatype, target_rank,</pre>			
37	<pre>target_disp, op, win, ierror) TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: origin_addr</pre>					
38	TYPE(*), DIMENSION(), INTENT(IN), ASTNCHRONOUS :: OFIGIN_addr TYPE(*), DIMENSION(), ASYNCHRONOUS :: result_addr					
39 40	TYPE(MPI_Datatype), INTENT(IN) :: datatype					
41	INTEGER, INTENT(IN) :: target_rank					
42	INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp TYPE(MPI_Op), INTENT(IN) :: op					
43	TYPE(MPI_OP), INTENT(IN) :: win					
44 45	INTEG	ER, OPTIONAL, INTENT(OUT)	) :: ierror			
46	Fortran b	inding				
47	MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,					
48		TARGET_DISP, OP, WIN	, IERROR)			

<type> ORIGIN_ADDR(*), RESULT_ADDR(*) INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP</type>				
Accumulate one element of type datatype from the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation. The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be specified as op; user-defined functions cannot be used. The datatype argument must be a predefined datatype. The operation is executed atomically.				
Compare a	nd Swap Function		13 14	
Another useful operation is an atomic compare and swap where the value at the origin is compared to the value at the target, which is atomically replaced by a third value only if the values at origin and target are equal.				
MPI_COM	PARE_AND_SWAP(origin_add target_rank, target_disp,	r, compare_addr, result_addr, datatype, win)	20 21	
IN	origin_addr	initial address of buffer (choice)	22	
IN	compare_addr	initial address of compare buffer (choice)	23 24	
OUT	result_addr	initial address of result buffer (choice)	25	
IN	datatype	datatype of the element in all buffers (handle)	26	
IN	target_rank	rank of target (non-negative integer)	27	
IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)	28 29 30	
IN	win	window object (handle)	31 32	
C binding int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)			33 34 35 36 37	
Fortran 2	2008 binding		38	
MPI_Compa		<pre>compare_addr, result_addr, datatype,</pre>	39	
	target_rank, target_	-	40	
TYPE	(*), DIMENSIUN(), INTEN compare_addr	C(IN), ASYNCHRONOUS :: origin_addr,	41 42	
TYPE	(*), DIMENSION(), ASYNCH	IRONOUS :: result_addr	43	
TYPE	(MPI_Datatype), INTENT(IN)	:: datatype	44	
	ER, INTENT(IN) :: target		45	
	GER(KIND=MPI_ADDRESS_KIND) (MPI_Win), INTENT(IN) :: v	, INTENT(IN) :: target_disp	$46 \\ 47$	
	ER, OPTIONAL, INTENT(UN) :: V		48	

1	Fortran binding
2	MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
3	TARGET_RANK, TARGET_DISP, WIN, IERROR)
4	<type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)</type>
5	INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
6	INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP

7 This function compares one element of type datatype in the compare buffer 8 compare\_addr with the buffer at offset target\_disp in the target window specified by 9 target\_rank and win and replaces the value at the target with the value in the origin buffer 10 origin\_addr if the compare buffer and the target buffer are identical. The original value at 11 the target is returned in the buffer result\_addr. The parameter datatype must belong to 12one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, 13 Multi-language types, or Byte as specified in Section 6.9.2. The origin and result buffers 14(origin\_addr and result\_addr) must be disjoint. 15

16

## 17 12.3.5 Request-based RMA Communication Operations

<sup>18</sup> Request-based RMA communication operations allow the user to associate a request handle <sup>19</sup> with the RMA operations and test or wait for the completion of these requests using the <sup>20</sup> functions described in Section 3.7.3. Request-based RMA operations are only valid within <sup>21</sup> a passive target epoch (see Section 12.5).

<sup>22</sup> Upon returning from a completion call in which an RMA operation completes, all fields <sup>23</sup> of the status object, if any, and the results of status query functions (e.g.,

<sup>24</sup> MPI\_GET\_COUNT) are undefined with the exception of MPI\_ERROR if appropriate (see <sup>25</sup> Section 3.2.5). It is valid to mix different request types (e.g., any combination of RMA <sup>26</sup> requests, collective requests, I/O requests, generalized requests, or point-to-point requests) <sup>27</sup> in functions that enable multiple completions (e.g., MPI\_WAITALL). It is erroneous to call <sup>28</sup> MPI\_REQUEST\_FREE or MPI\_CANCEL for a request associated with an RMA operation. <sup>29</sup> RMA requests are not persistent.

<sup>30</sup> The end of the epoch, or explicit bulk synchronization using

<sup>31</sup> MPI\_WIN\_FLUSH, MPI\_WIN\_FLUSH\_ALL, MPI\_WIN\_FLUSH\_LOCAL, or

MPI\_WIN\_FLUSH\_LOCAL\_ALL, also indicates completion of the RMA operations. How ever, users must still wait or test on the request handle to allow the MPI implementation to
 clean up any resources associated with these requests; in such cases the wait operation will
 complete locally.

- 36
- 37
- 38
- 39
- 40 41
- 42

- 44
- 45
- 46
- 47 48

MPI_RPUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, <sup>1</sup> target_count, target_datatype, win, request) <sup>2</sup>				
IN	origin_addr	initial address of origin buffer (choice)	3	
IN	origin_count	number of entries in origin buffer (non-negative integer)	4 5 6	
IN	origin_datatype	datatype of each entry in origin buffer (handle)	7	
IN	target_rank	rank of target (non-negative integer)	8	
IN	target_disp	displacement from start of window to target buffer (non-negative integer)	9 10 11	
IN	target_count	number of entries in target buffer (non-negative integer)	12 13	
IN	target_datatype	datatype of each entry in target buffer (handle)	14	
IN	win	window object used for communication (handle)	15 16	
OUT	request	RMA request (handle)	17	
			18	
C bindi	ng		19 20	
<pre>int MPI_Rput(const void *origin_addr, int origin_count,</pre>				
MPI_Datatype origin_datatype, int target_rank,				
MPI_Aint target_disp, int target_count,			22 23	
	• -	get_datatype, MPI_Win win,	24	
MPI_Request *request)			25	
<pre>int MPI_Rput_c(const void *origin_addr, MPI_Count origin_count,</pre>			26	
MPI_Datatype origin_datatype, int target_rank,			27	
MPI_Aint target_disp, MPI_Count target_count,			28	
MPI_Datatype target_datatype, MPI_Win win,			29	
	MPI_Request *req	uest)	30	
Fortran	2008 binding		31	
MPI_Rput	c(origin_addr, origin_	count, origin_datatype, target_rank,	32 33	
		get_count, target_datatype, win, request,	34	
	ierror)		35	
		NTENT(IN), ASYNCHRONOUS :: origin_addr	36	
		<pre>igin_count, target_rank, target_count T(IN) :: origin_datatype, target_datatype</pre>	37	
	<b>U</b> 1	KIND), INTENT(IN) :: target_datatype	38	
	EGER(KIND-MFI_ADDRESS_ E(MPI_Win), INTENT(IN)	<b>o i</b>	39	
	E(MPI_Request), INTENT		40	
	EGER, OPTIONAL, INTENT		41	
			42	
MP1_Rput		count, origin_datatype, target_rank,	43	
	<pre>ierror) !(_c)</pre>	get_count, target_datatype, win, request,	44 45	
түрг		NTENT(IN), ASYNCHRONOUS :: origin_addr	45 46	
		ND), INTENT(IN) :: origin_count, target_count	40	
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype       48				

1 2 3 4	INTE TYPE TYPE	(MPI_Win), INTENT(IN) :: (MPI_Request), INTENT(OU	D), INTENT(IN) :: target_disp win T) :: request		
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
6	Fortran	binding			
7		6	NT, ORIGIN_DATATYPE, TARGET_RANK,		
8 9		-	COUNT, TARGET_DATATYPE, WIN, REQUEST,		
10		IERROR)			
11	<typ< th=""><th>e&gt; ORIGIN_ADDR(*)</th><th></th></typ<>	e> ORIGIN_ADDR(*)			
12	INTE		_DATATYPE, TARGET_RANK, TARGET_COUNT,		
13			IN, REQUEST, IERROR		
14	INTE	GER(KIND=MPI_ADDRESS_KIN	D) TARGET_DISP		
15	MPI_	RPUT is similar to MPI_PUT	(Section 12.3.1), except that it allocates a commu-		
16		× 0	it with the request handle (the argument $\ensuremath{request}).$		
17 18	-		ration (i.e., after the corresponding test or wait) in-		
19			update the locations in the origin buffer. It does		
20			at the target window. If remote completion is re- FLUSH_ALL, MPI_WIN_UNLOCK, or		
21		_UNLOCK_ALL can be used.			
22					
23		<b>T</b> ( · · · · · · · · · · · · · · · · · ·			
24 25	MPI_RGE	I (origin_addr, origin_count, or target_count, target_da	rigin_datatype, target_rank, target_disp, tatype, win, request)		
26	OUT	origin_addr	initial address of origin buffer (choice)		
27 28 29	IN	origin_count	number of entries in origin buffer (non-negative integer)		
30	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
31	IN	target_rank	rank of target (non-negative integer)		
32 33 34	IN	target_disp	displacement from window start to the beginning of the target buffer (non-negative integer)		
35 36	IN	target_count	number of entries in target buffer (non-negative integer)		
37	IN	target_datatype	datatype of each entry in target buffer (handle)		
38 39	IN	win	window object used for communication (handle)		
40	OUT	request	RMA request (handle)		
41					
42	C bindir	lg			
43	int MPI_	Rget(void *origin_addr, :	-		
44			_datatype, int target_rank,		
45 46		-	p, int target_count, _datatype, MPI_Win win,		
47		MPI_Datatype target MPI_Request *reques	• •		
48		<u>-</u>	-,		

```
1
int MPI_Rget_c(void *origin_addr, MPI_Count origin_count,
                                                                                   \mathbf{2}
              MPI_Datatype origin_datatype, int target_rank,
                                                                                   3
              MPI_Aint target_disp, MPI_Count target_count,
                                                                                   4
              MPI_Datatype target_datatype, MPI_Win win,
              MPI_Request *request)
                                                                                   5
                                                                                   6
Fortran 2008 binding
                                                                                   7
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                    8
              target_disp, target_count, target_datatype, win, request,
                                                                                   9
              ierror)
                                                                                   10
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                   11
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                   12
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                   13
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                   14
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   17
                                                                                   18
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                   19
              target_disp, target_count, target_datatype, win, request,
                                                                                   20
              ierror) !(_c)
                                                                                   21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                   22
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
                                                                                   23
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                   24
    INTEGER, INTENT(IN) :: target_rank
                                                                                   25
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                   26
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   27
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   29
Fortran binding
                                                                                   30
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
                                                                                   31
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
                                                                                   32
              IERROR)
                                                                                   33
    <type> ORIGIN_ADDR(*)
                                                                                   34
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
                                                                                   35
               TARGET_DATATYPE, WIN, REQUEST, IERROR
                                                                                   36
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
                                                                                   37
                                                                                   38
    MPI_RGET is similar to MPI_GET (Section 12.3.2), except that it allocates a commu-
                                                                                   39
```

MPI\_RGET is similar to MPI\_GET (Section 12.3.2), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI\_RGET operation indicates that the data is available in the origin buffer. If origin\_addr points to memory attached to a window, then the data becomes available in the private copy of this window.

587

47 48

40

41

42

```
588
                                        CHAPTER 12. ONE-SIDED COMMUNICATIONS
1
     MPI_RACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp,
\mathbf{2}
                    target_count, target_datatype, op, win, request)
3
       IN
                origin_addr
                                            initial address of buffer (choice)
4
       IN
                origin_count
                                            number of entries in buffer (non-negative integer)
5
6
                origin_datatype
                                            datatype of each entry in origin buffer (handle)
       IN
7
       IN
                target_rank
                                            rank of target (non-negative integer)
8
       IN
                target_disp
                                            displacement from start of window to beginning of
9
                                            target buffer (non-negative integer)
10
11
       IN
                target_count
                                            number of entries in target buffer (non-negative
12
                                            integer)
13
       IN
                target_datatype
                                            datatype of each entry in target buffer (handle)
14
       IN
                                            reduce operation (handle)
                ор
15
16
       IN
                win
                                            window object (handle)
17
       OUT
                request
                                            RMA request (handle)
18
19
     C binding
20
     int MPI_Raccumulate(const void *origin_addr, int origin_count,
21
                    MPI_Datatype origin_datatype, int target_rank,
22
                    MPI_Aint target_disp, int target_count,
23
                    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
24
                    MPI_Request *request)
25
26
     int MPI_Raccumulate_c(const void *origin_addr, MPI_Count origin_count,
27
                    MPI_Datatype origin_datatype, int target_rank,
                    MPI_Aint target_disp, MPI_Count target_count,
28
29
                    MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
30
                    MPI_Request *request)
^{31}
     Fortran 2008 binding
32
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
33
                    target_disp, target_count, target_datatype, op, win, request,
34
                    ierror)
35
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
36
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
37
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
38
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
39
         TYPE(MPI_Op), INTENT(IN) :: op
40
         TYPE(MPI_Win), INTENT(IN) :: win
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
45
                    target_disp, target_count, target_datatype, op, win, request,
46
                    ierror) !(_c)
47
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
48
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
```

TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype	1
INTEGER, INTENT(IN) :: target_rank	2
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp	3
TYPE(MPI_Op), INTENT(IN) :: op	4
TYPE(MPI_Win), INTENT(IN) :: win	5
TYPE(MPI_Request), INTENT(OUT) :: request	6
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	7
Fortron hinding	8
Fortran binding	9
MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	10
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	11
IERROR)	12
<type> ORIGIN_ADDR(*)</type>	13
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	14
TARGET_DATATYPE, OP, WIN, REQUEST, IERROR	15
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	16
MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section $12.3.4$ ), except that	17
	18

it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI\_RACCUMULATE operation indicates that the origin buffer is free to be updated. It does not indicate that the operation has completed at the target window.

1 2	MPI_RGET	· -	origin_count, origin_datatype, result_addr, ype, target_rank, target_disp, target_count,		
3		target_datatype, op, win,			
4 5	IN	origin_addr	initial address of buffer (choice)		
6 7	IN	origin_count	number of entries in origin buffer (non-negative integer)		
8	IN	origin_datatype	datatype of each entry in origin buffer (handle)		
9 10	OUT	result_addr	initial address of result buffer (choice)		
10 11 12	IN	result_count	number of entries in result buffer (non-negative integer)		
13	IN	result_datatype	datatype of entries in result buffer (handle)		
14	IN	target_rank	rank of target (non-negative integer)		
15 16 17	IN	target_disp	displacement from start of window to beginning of target buffer (non-negative integer)		
18 19	IN	target_count	number of entries in target buffer (non-negative integer)		
20 21	IN	target_datatype	datatype of each entry in target buffer (handle)		
21	IN	ор	reduce operation (handle)		
23	IN	win	window object (handle)		
24 25	OUT	request	RMA request (handle)		
26 27 28 29 30 31 32 33	C binding int MPI_Rget_accumulate(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, void *result_addr, int result_count, MPI_Datatype result_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win, MPI_Bequest *request)				
34 35 36 37 38 39 40	<pre>int MPI_Rget_accumulate_c(const void *origin_addr, MPI_Count origin_count,</pre>				
41 42 43 44 45 46 47 48	MPI_Rget_	<pre>result_addr, result_d target_disp, target_d ierror) *), DIMENSION(), INTENT</pre>	origin_count, origin_datatype, count, result_datatype, target_rank, count, target_datatype, op, win, request, C(IN), ASYNCHRONOUS :: origin_addr count, result_count, target_rank,		

```
TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                   1
                                                                                   2
              target_datatype
                                                                                   3
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                   4
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   5
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   6
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   9
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
                                                                                   10
              result_addr, result_count, result_datatype, target_rank,
                                                                                   11
              target_disp, target_count, target_datatype, op, win, request,
                                                                                   12
              ierror) !(_c)
                                                                                   13
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                   14
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
                                                                                   15
              target_count
                                                                                   16
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
                                                                                   17
              target_datatype
                                                                                   18
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
                                                                                   19
    INTEGER, INTENT(IN) :: target_rank
                                                                                   20
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                   21
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   22
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   23
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   ^{24}
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   25
                                                                                   26
Fortran binding
                                                                                   27
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
                                                                                   28
              RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
                                                                                   29
              TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
                                                                                   30
              IERROR)
                                                                                   31
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
                                                                                   32
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
                                                                                   33
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
                                                                                   34
              IERROR
                                                                                   35
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
                                                                                   36
```

MPI\_RGET\_ACCUMULATE is similar to MPI\_GET\_ACCUMULATE (Section 12.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI\_RGET\_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

# 12.4 Memory Model

The memory semantics of RMA are best understood by using the concept of *public* and <sup>46</sup> *private* window copies. We assume that systems have a public memory region that is <sup>47</sup> addressable by all processes (e.g., the shared memory in shared memory machines or the <sup>48</sup>

37

38

39

40

41

42 43

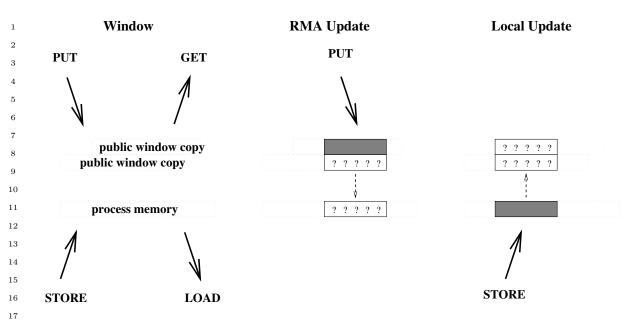


Figure 12.1: Schematic description of the public/private window operations in the MPI\_WIN\_SEPARATE memory model for two overlapping windows

19 20

18

21exposed main memory in distributed memory machines). In addition, most machines have 22fast private buffers (e.g., transparent caches or explicit communication buffers) local to each 23process where copies of data elements from the main memory can be stored for faster access.  $^{24}$ Such buffers are either coherent, i.e., all updates to main memory are reflected in all private 25copies consistently, or noncoherent, i.e., conflicting accesses to main memory need to be 26synchronized and updated in all private copies explicitly. Coherent systems allow direct 27updates to remote memory without any participation of the remote side. Noncoherent 28systems, however, need to call RMA functions in order to reflect updates to the public 29window in their private memory. Thus, in coherent memory, the public and the private 30 window are identical while they remain logically separate in the noncoherent case. MPI  $^{31}$ thus differentiates between two **memory models** called **RMA unified**, if public and private 32 window are logically identical, and **RMA** separate, otherwise.

33 In the RMA separate model, there is only one instance of each variable in process 34memory, but a distinct *public* copy of the variable for each window that contains it. A load 35 accesses the instance in process memory (this includes MPI sends). A local store accesses 36 and updates the instance in process memory (this includes MPI receives), but the update 37 may affect other public copies of the same locations. A get on a window accesses the public 38 copy of that window. A put or accumulate on a window accesses and updates the public 39 copy of that window, but the update may affect the private copy of the same locations 40in process memory, and public copies of other overlapping windows. This is illustrated in  $^{41}$ Figure **12.1**.

<sup>42</sup> In the RMA unified model, public and private copies are identical and updates via put <sup>43</sup> or accumulate calls are eventually observed by load operations without additional RMA <sup>44</sup> calls. A store access to a window is eventually visible to remote get or accumulate calls <sup>45</sup> without additional RMA calls. These stronger semantics of the RMA unified model allow <sup>46</sup> the user to omit some synchronization calls and potentially improve performance.

47 48

Advice to users. If accesses in the RMA unified model are not synchronized (with

locks or flushes, see Section 12.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

The memory model for a particular RMA window can be determined by accessing the attribute MPI\_WIN\_MODEL. If the memory model is the unified model, the value of this attribute is MPI\_WIN\_UNIFIED; otherwise, the value is MPI\_WIN\_SEPARATE.

# 12.5 Synchronization Calls

RMA communications fall in two categories:

- active target communication, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.
- passive target communication, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI\_PUT, MPI\_GET or MPI\_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1 2

3

4

5

6

 $\overline{7}$ 

8

9 10 11

12 13

14

15

16

17

18

19 20

21

22

23

 $^{24}$ 

25

26

27 28

29

30

 $^{31}$ 

32 33

34

35

36 37

38 39

40 41

42

43 44

 $45 \\ 46$ 

47

- 1. The MPI\_WIN\_FENCE collective synchronization call supports a simple synchroniza-2 tion pattern that is often used in parallel computations: namely a loosely-synchronous 3 model, where global computation phases alternate with global communication phases. 4 This mechanism is most useful for loosely synchronous algorithms where the graph 5of communicating processes changes very frequently, or where each process communi-6 cates with many others.
  - This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI\_WIN\_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.
- 2. The four functions MPI\_WIN\_START, MPI\_WIN\_COMPLETE, MPI\_WIN\_POST, and 13 14MPI\_WIN\_WAIT can be used to restrict synchronization to the minimum: only pairs 15of communicating processes synchronize, and they do so only when a synchronization 16is needed to order correctly RMA accesses to a window with respect to local accesses 17 to that same window. This mechanism may be more efficient when each process 18 communicates with few (logical) neighbors, and the communication graph is fixed or 19 changes infrequently.
- 20These calls are used for active target communication. An access epoch is started 21at the origin process by a call to MPI\_WIN\_START and is terminated by a call to 22MPI\_WIN\_COMPLETE. The start call has a group argument that specifies the group 23of target processes for that epoch. An exposure epoch is started at the target process 24by a call to MPI\_WIN\_POST and is completed by a call to MPI\_WIN\_WAIT. The post 25call has a group argument that specifies the set of origin processes for that epoch. 26
- 273. Finally, shared lock access is provided by the functions MPI\_WIN\_LOCK,
  - MPI\_WIN\_LOCK\_ALL, MPI\_WIN\_UNLOCK, and MPI\_WIN\_UNLOCK\_ALL.
- 29MPI\_WIN\_LOCK and MPI\_WIN\_UNLOCK also provide exclusive lock capability. Lock 30 synchronization is useful for MPI applications that emulate a shared memory model 31via MPI calls; e.g., in a "bulletin board" model, where processes can, at random times, 32 access or update different parts of the bulletin board.
- 33 These four calls provide passive target communication. An access epoch is started 34by a call to MPI\_WIN\_LOCK or MPI\_WIN\_LOCK\_ALL and terminated by a call to 35 MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL, respectively. 36
- 37 Figure 12.2 illustrates the general synchronization pattern for active target communi-38cation. The synchronization between **post** and **start** ensures that the put call of the origin 39 process does not start until the target process exposes the window (with the **post** call); 40the target process will expose the window only after preceding local accesses to the window  $^{41}$ have completed. The synchronization between complete and wait ensures that the put call 42of the origin process completes before the window is unexposed (with the wait call). The 43target process will execute following local accesses to the target window only after the wait 44returned.
- 45Figure 12.2 shows operations occurring in the natural temporal order implied by the 46synchronizations: the post occurs before the matching start, and complete occurs be-47fore the matching wait. However, such strong synchronization is more than needed for 48

1

7

8

9

10

11

12

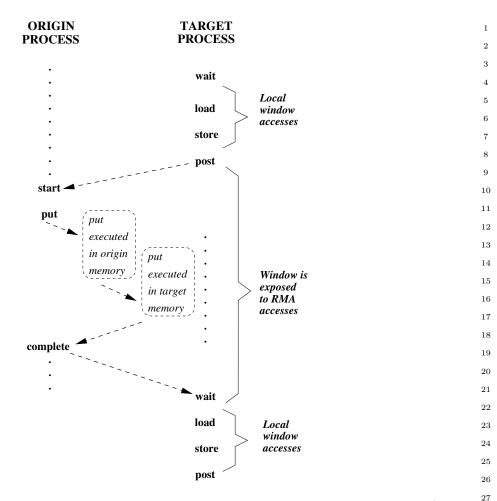


Figure 12.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

correct ordering of window accesses. The semantics of MPI calls allow **weak synchroniza-tion**, as illustrated in Figure 12.3. The access to the target window is delayed until the window is exposed, after the post. However the **start** may complete earlier; the **put** and **complete** may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 12.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The lock and unlock calls ensure that the two RMA accesses do not occur concurrently. However, they do *not* ensure that the put by origin 1 will precede the get by origin 2.

*Rationale.* RMA does not define fine-grained mutexes in memory (only logical coarsegrained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End of rationale.*)

```
ORIGIN
                                                     TARGET
1
                             PROCESS
                                                     PROCESS
2
3
                                 •
                                                       wait
4
5
                               start
                                                                  Local
                                                       load
                                                                  window
6
                                                                  accesses
                                put
7
                                                       store
8
                                     put
                                                       post
9
                                     executed
10
                                     in origin
11
                                                                  Window is
12
                                     memory
                                               put
                                                                  exposed
                                                                  to RMA
13
                                               executed
                                                                  accesses
14
                                               in target
15
                                               memory
                             complete
16
17
                                                       wait
18
19
                                                                   Local
                                                       load
                                                                   window
20
                                                                   accesses
                                                       store
21
22
                                                       post
23
24
      Figure 12.3: Active target communication, with weak synchronization. Dashed arrows
25
      represent synchronizations (ordering of events).
26
27
      12.5.1 Fence
28
29
30
      MPI_WIN_FENCE(assert, win)
31
32
        IN
                                                program assertion (integer)
                  assert
33
        IN
                  win
                                                window object (handle)
34
35
      C binding
36
      int MPI_Win_fence(int assert, MPI_Win win)
37
38
      Fortran 2008 binding
39
      MPI_Win_fence(assert, win, ierror)
40
          INTEGER, INTENT(IN) :: assert
41
          TYPE(MPI_Win), INTENT(IN) :: win
42
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
      Fortran binding
44
      MPI_WIN_FENCE(ASSERT, WIN, IERROR)
45
          INTEGER ASSERT, WIN, IERROR
46
47
          The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call
48
```

CHAPTER 12. ONE-SIDED COMMUNICATIONS

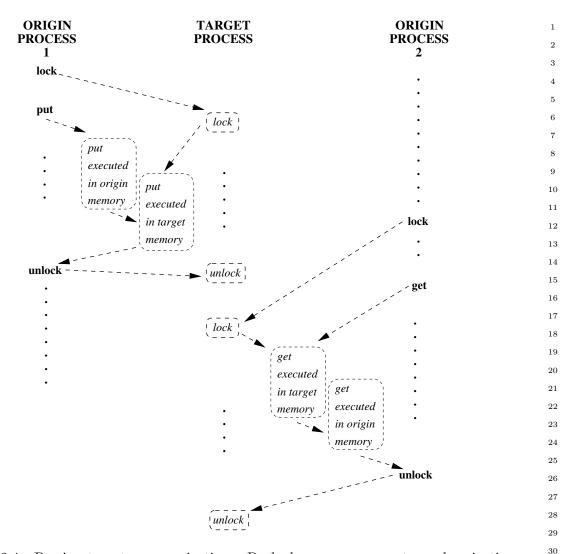


Figure 12.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI\_WIN\_FENCE has been called by the target process.

The call completes an RMA access epoch if it was preceded by another fence call and 39 the local process issued RMA communication calls on win between these two calls. The call 40 completes an RMA exposure epoch if it was preceded by another fence call and the local 41 window was the target of RMA accesses between these two calls. The call starts an RMA 42access epoch if it is followed by another fence call and by RMA communication calls issued 43 between these two fence calls. The call starts an exposure epoch if it is followed by another 44fence call and the local window is the target of RMA accesses between these two fence calls. 45Thus, the fence call is equivalent to calls to a subset of post, start, complete, wait. 46

A fence call usually entails a barrier synchronization: a process completes a call to 47 MPI\_WIN\_FENCE only after all other processes in the group entered their matching call. 48

 $^{31}$ 

32 33

34

35

36

37

1 However, a call to MPI\_WIN\_FENCE that is known not to end any epoch (in particular, a  $\mathbf{2}$ call with assert equal to MPI\_MODE\_NOPRECEDE) does not necessarily act as a barrier. 3 The assert argument is used to provide assertions on the context of the call that may 4 be used for various optimizations. This is described in Section 12.5.5. A value of assert =  $\mathbf{5}$ 0 is always valid. 6 Advice to users. Calls to MPI\_WIN\_FENCE should both precede and follow calls to 7 RMA communication functions that are synchronized with fence calls. (End of advice 8 to users.) 9 10 12.5.2 General Active Target Synchronization 11 1213 14MPI\_WIN\_START(group, assert, win) 15IN group of target processes (handle) group 16IN assert program assertion (integer) 1718 IN window object (handle) win 19 20C binding 21int MPI\_Win\_start(MPI\_Group group, int assert, MPI\_Win win) 22Fortran 2008 binding 23MPI\_Win\_start(group, assert, win, ierror) 24TYPE(MPI\_Group), INTENT(IN) :: group 25INTEGER, INTENT(IN) :: assert 26TYPE(MPI\_Win), INTENT(IN) :: win 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2829Fortran binding 30 MPI\_WIN\_START(GROUP, ASSERT, WIN, IERROR)  $^{31}$ INTEGER GROUP, ASSERT, WIN, IERROR 32 Starts an RMA access epoch for win. RMA calls issued on win during this epoch must 33 access only windows at processes in group. Each process in group must issue a matching 34 call to MPI\_WIN\_POST. RMA accesses to each target window will be delayed, if necessary, 35 until the target process executed the matching call to MPI\_WIN\_POST. MPI\_WIN\_START 36 is allowed to block until the corresponding MPI\_WIN\_POST calls are executed, but is not 37 required to. 38 The assert argument is used to provide assertions on the context of the call that may 39 be used for various optimizations. This is described in Section 12.5.5. A value of assert =40 0 is always valid. 41 4243 MPI\_WIN\_COMPLETE(win) 44 IN window object (handle) win 454647C binding 48int MPI\_Win\_complete(MPI\_Win win)

#### Fortran 2008 binding

```
MPI_Win_complete(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### Fortran binding

MPI\_WIN\_COMPLETE(WIN, IERROR) INTEGER WIN, IERROR

Completes an RMA access epoch on win started by a call to MPI\_WIN\_START. All RMA communication calls issued on win during this epoch will have completed at the origin when the call returns.

MPI\_WIN\_COMPLETE enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin.

Consider the sequence of calls in the example below.

### Example 12.4 Use of MPI\_WIN\_START and MPI\_WIN\_COMPLETE.

```
MPI_Win_start(group, flag, win);
MPI_Put(..., win);
MPI_Win_complete(win);
```

The call to MPI\_WIN\_COMPLETE does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to MPI\_WIN\_START has matched a call to MPI\_WIN\_POST by the target process. This still leaves much choice to implementors. The call to MPI\_WIN\_START can block until the matching call to MPI\_WIN\_POST occurs at all target processes. One can also have implementations where the call to MPI\_WIN\_START is nonblocking, but the call to MPI\_PUT blocks until the matching call to MPI\_WIN\_START is nonblocking, but the call to MPI\_PUT blocks until the matching call to MPI\_WIN\_POST occurs; or implementations where the first two calls are nonblocking, but the call to MPI\_WIN\_COMPLETE blocks until the call to MPI\_WIN\_POST occurred; or even implementations where all three calls can complete before any target process has called MPI\_WIN\_POST—the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI\_WIN\_POST is issued, the sequence above must complete, without further dependencies.

MPI\_WIN\_POST(group, assert, win)

IN	group	group of origin processes (handle)
IN	assert	program assertion (integer)
IN	win	window object (handle)
C bind	ling	

int MPI\_Win\_post(MPI\_Group group, int assert, MPI\_Win win)

# Fortran 2008 binding MPI\_Win\_post(group, assert, win, ierror) TYPE(MPI\_Group), INTENT(IN) :: group

 $\mathbf{2}$ 

 $45 \\ 46$ 

1			IN) :: assert NTENT(IN) :: wi	a	
3	INTEGER, OPTIONAL, INTENT(OUT) :: ierror				
4 5 6 7	MPI_WIN_	Fortran binding MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR) INTEGER GROUP, ASSERT, WIN, IERROR			
8 9 10 11	Starts an RMA exposure epoch for the local window associated with win. Only processes in group should access the window with RMA calls on win during this epoch. Each process in group must issue a matching call to MPI_WIN_START. MPI_WIN_POST does not block.				
12 13	MPI WIN	_WAIT(win)			
14 15	IN	win		vindow object (handle)	
16 17 18	C bindin int MPI_	eg Win_wait(MPI	_Win win)		
18	Fortran 2	2008 binding			
20		wait(win, ie:			
21 22			NTENT(IN) :: wi L, INTENT(OUT)		
23 24	Fortran	binding			
25 26	MPI_WIN_WAIT(WIN, IERROR) INTEGER WIN, IERROR				
27 28 29 30 31 32	Completes an RMA exposure epoch started by a call to MPI_WIN_POST on win. This call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.				
33 34 35 36 37	Figure 12.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start,				
38 39 40	put or cor	npiete cans ma	ly occur anead of	the matching post calls.	
41	MPI_WIN	_TEST(win, fla	g)		
42	IN	win		vindow object (handle)	
43 44	OUT	flag	1	success flag (logical)	
45 46 47 48	C bindin int MPI_	-	_Win win, int *	flag)	

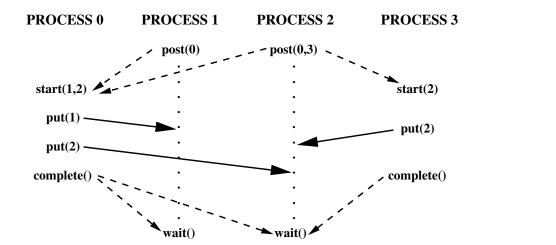


Figure 12.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

### Fortran 2008 binding

```
MPI_Win_test(win, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### Fortran binding

MPI\_WIN\_TEST(WIN, FLAG, IERROR) INTEGER WIN, IERROR LOGICAL FLAG

This is the nonblocking version of MPI\_WIN\_WAIT. It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding MPI\_WIN\_POST call have been completed as signalled by matching MPI\_WIN\_COMPLETE calls, and flag = false otherwise. In the former case MPI\_WIN\_WAIT would have returned immediately. The effect of return of MPI\_WIN\_TEST with flag = true is the same as the effect of a return of MPI\_WIN\_WAIT. If flag = false is returned, then the call has no visible effect.

MPI\_WIN\_TEST should be invoked only where MPI\_WIN\_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait calls can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

# MPI\_WIN\_POST(group,0,win) initiates a nonblocking send with tag tag0 to each process in group, using wincomm. There is no need to wait for the completion of these sends.

MPI\_WIN\_START(group,0,win) initiates a nonblocking receive with tag tag0 from each process in group, using wincomm. An RMA access to a window in target process i is delayed until the receive from i is completed.

- **MPI\_WIN\_COMPLETE(win)** initiates a nonblocking send with tag tag1 to each process in the group of the preceding start call. No need to wait for the completion of these sends.
  - **MPI\_WIN\_WAIT(win)** initiates a nonblocking receive with tag tag1 from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice versa.

Rationale. The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

Advice to users. Assume a communication pattern that is represented by a directed graph  $G = \langle V, E \rangle$ , where  $V = \{0, ..., n-1\}$  and  $ij \in E$  if origin process *i* accesses the window at target process *j*. Then each process *i* issues a call to MPI\_WIN\_POST(*ingroup*<sub>i</sub>, ...), followed by a call to

MPI\_WIN\_START( $outgroup_i,...$ ), where  $outgroup_i = \{j : ij \in E\}$  and  $ingroup_i = \{j : ji \in E\}$ . A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

```
12.5.3 Lock
```

```
MPI_WIN_LOCK(lock_type, rank, assert, win)
```

```
37
       IN
                  lock_type
                                               either MPI_LOCK_EXCLUSIVE or
38
                                               MPI_LOCK_SHARED (state)
39
       IN
                                               rank of locked window (non-negative integer)
                  rank
40
41
       IN
                  assert
                                               program assertion (integer)
42
       IN
                                               window object (handle)
                 win
43
44
     C binding
45
     int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
46
47
     Fortran 2008 binding
```

<sup>48</sup> MPI\_Win\_lock(lock\_type, rank, assert, win, ierror)

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

16

17

18

19

20

26

27

28

29

30

31 32

INT	EGER, INTENT(IN)	:: lock_type, rank, assert	1
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror			2
INT	EGER, OPTIONAL,	INTENT(OUT) :: ierror	3
Fortran binding			4 5
	MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)		
	INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR		
Ct			7 8
		poch. The window at the process with rank rank can be accessed	9
by RMA operations on win during that epoch. Multiple RMA access epochs (with calls to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a			0
	process.		.1
unierent	process.	1	2
			.3
MPI_WI	N_LOCK_ALL(asser	t, win) 1	.4
IN	assert	program assertion (integer)	5
IN	win	window object (handle)	.6
		1	7
C bindi	no		.8
	0	tassert MPI Win Win)	.9
		· · ·	20 21
	2008 binding		22
	_lock_all(assert	, win, ierror)	23
INTEGER, INTENT(IN) :: assert TYPE(MPI Win) INTENT(IN) win			24
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror			25
TNI	LOLIT, OF FIONAL,		26
Fortran binding			27
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)			28
INT	EGER ASSERT, WIN	, IERROR 2	29
Star	rts an RMA access	epoch to all processes in win, with a lock type of <sup>3</sup>	80
MPI_LOO	CK_SHARED. During	the epoch, the calling process can access the window memory on	31
all proce	sses in win by using	KMA operations. A window locked with MPI_VVIN_LOCK_ALL	32
must be	unlocked with $MP$	1_VVIN_UNLOCK_ALL. This routine is not collective—the ALL	33
refers to	a lock on all memb	Ders of the group of the window.	34
4.0	lvice to users. Th		35 36
			37
			88
			39
1	Υ.		10
		4	1
MPI WI	N_UNLOCK(rank, v	<sup>4</sup>	12
	rank	4	13
IN			4
IN	win		15
			16
C binding			17
int MPI_Win_unlock(int rank, MPI_Win win) 48			ð

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Win_unlock(rank, win, ierror)
3
          INTEGER, INTENT(IN) :: rank
4
          TYPE(MPI_Win), INTENT(IN) :: win
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_WIN_UNLOCK(RANK, WIN, IERROR)
8
          INTEGER RANK, WIN, IERROR
9
10
          Completes an RMA access epoch started by a call to MPI_WIN_LOCK on window win.
11
     RMA operations issued during this period will have completed both at the origin and at the
12
     target when the call returns.
13
14
     MPI_WIN_UNLOCK_ALL(win)
15
16
       IN
                                              window object (handle)
                 win
17
18
     C binding
19
     int MPI_Win_unlock_all(MPI_Win win)
20
21
     Fortran 2008 binding
     MPI_Win_unlock_all(win, ierror)
22
          TYPE(MPI_Win), INTENT(IN) :: win
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
     Fortran binding
26
     MPI_WIN_UNLOCK_ALL(WIN, IERROR)
27
          INTEGER WIN, IERROR
28
          Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL on
29
     window win. RMA operations issued during this epoch will have completed both at the
30
     origin and at the target when the call returns.
^{31}
32
          Locks are used to protect accesses to the locked target window effected by RMA calls
33
     issued between the lock and unlock calls, and to protect load/store accesses to a locked local
34
     or shared memory window executed between the lock and unlock calls. Accesses that are
35
     protected by an exclusive lock will not be concurrent at the window site with other accesses
36
     to the same window that are lock protected. Accesses that are protected by a shared lock
37
     will not be concurrent at the window site with accesses protected by an exclusive lock to
38
     the same window.
39
          It is erroneous to have a window locked and exposed (in an exposure epoch) concur-
40
     rently. For example, a process may not call MPI_WIN_LOCK to lock a target window if
41
     the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it
42
     is erroneous to call MPI_WIN_POST while the local window is locked.
43
44
           Rationale.
                        An alternative is to require MPI to enforce mutual exclusion between
45
           exposure epochs and locking periods. But this would entail additional overheads
46
           when locks or active target synchronization do not interact in support of those rare
47
           interactions between the two mechanisms. The programming style that we encourage
48
```

here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

Advice to users. Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (End of advice to users.)

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI\_ALLOC\_MEM (Section 9.2), MPI\_WIN\_ALLOCATE (Section 12.2.2), MPI\_WIN\_ALLOCATE\_SHARED (Section 12.2.3), or attached with MPI\_WIN\_ATTACH (Section 12.2.4). Locks can be used portably only in such memory.

*Rationale.* The implementation of passive target communication when memory is not shared may require an asynchronous software agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for third party communication in shared memory machines.

(End of rationale.)

Consider the sequence of calls in the example below.

Example 12.5 Use of MPI\_WIN\_LOCK and MPI\_WIN\_UNLOCK.

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win);
MPI_Put(..., rank, ..., win);
MPI_Win_unlock(rank, win);
```

The call to MPI\_WIN\_UNLOCK will not return until the put transfer has completed at the origin and at the target. This still leaves much freedom to implementors. The call to MPI\_WIN\_LOCK may block until an exclusive lock on the window is acquired; or, the first two calls may not block, while MPI\_WIN\_UNLOCK blocks until a lock is acquired—the update of the target window is then postponed until the call to MPI\_WIN\_UNLOCK occurs. However, if the call to MPI\_WIN\_LOCK is used to lock a local window, then the call must block until the lock is acquired, since the lock may protect local load/store accesses to the window issued after the lock call returns.

### 12.5.4 Flush and Sync

All flush and sync functions can be called only within passive target epochs.

MPI\_WIN\_FLUSH(rank, win)

IN	rank	rank of target window (non-negative integer)
IN	win	window object (handle)

C binding int MPI\_Win\_flush(int rank, MPI\_Win win)  $\mathbf{2}$ 

```
1
     Fortran 2008 binding
\mathbf{2}
     MPI_Win_flush(rank, win, ierror)
3
          INTEGER, INTENT(IN) :: rank
4
          TYPE(MPI_Win), INTENT(IN) :: win
5
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     Fortran binding
7
     MPI_WIN_FLUSH(RANK, WIN, IERROR)
8
          INTEGER RANK, WIN, IERROR
9
10
          MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling
11
     process to the target rank on the specified window. The operations are completed both at
12
     the origin and at the target.
13
14
     MPI_WIN_FLUSH_ALL(win)
15
16
       IN
                 win
                                             window object (handle)
17
18
     C binding
19
     int MPI_Win_flush_all(MPI_Win win)
20
     Fortran 2008 binding
21
     MPI_Win_flush_all(win, ierror)
22
          TYPE(MPI_Win), INTENT(IN) :: win
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
     Fortran binding
26
     MPI_WIN_FLUSH_ALL(WIN, IERROR)
27
          INTEGER WIN, IERROR
28
         All RMA operations issued by the calling process to any target on the specified window
29
     prior to this call and in the specified window will have completed both at the origin and at
30
     the target when this call returns.
^{31}
32
33
     MPI_WIN_FLUSH_LOCAL(rank, win)
34
       IN
35
                 rank
                                             rank of target window (non-negative integer)
36
       IN
                 win
                                             window object (handle)
37
38
     C binding
39
     int MPI_Win_flush_local(int rank, MPI_Win win)
40
41
     Fortran 2008 binding
42
     MPI_Win_flush_local(rank, win, ierror)
43
          INTEGER, INTENT(IN) :: rank
44
          TYPE(MPI_Win), INTENT(IN) :: win
45
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     Fortran binding
47
     MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
48
```

INTEGER	RANK,	WIN,	IERROR
---------	-------	------	--------

Locally completes at the origin all outstanding RMA operations initiated by the calling process to the target process specified by rank on the specified window. For example, after this routine completes, the user may reuse any buffers provided to put, get, or accumulate operations.

MPI_WIN_FLUSH_LOCAL_ALL(win)			
IN	win	window object (handle)	9
	vviii	whitew object (handle)	10
C hindin			11
C bindin	₽ Win_flush_local_all(MPI_W	inin)	12
IIIC MFI_	win_iiusn_iocai_aii(MFi_w	111 W117	13
Fortran	2008 binding		14 15
MPI_Win_flush_local_all(win, ierror)			
TYPE(MPI_Win), INTENT(IN) :: win			
INTE	GER, OPTIONAL, INTENT(OUT	) :: ierror	17 18
Fortran	binding		18
	FLUSH_LOCAL_ALL(WIN, IERR	(au	20
	GER WIN, IERROR		20
	-		22
All RMA operations issued to any target prior to this call in this window will have			
completed	l at the origin when MPI_WIN	_FLUSH_LOCAL_ALL returns.	23 24
			25
	_SYNC(win)		26
	( ),		27
IN	win	window object (handle)	28
			29
C bindin	0		30
int MPI_	Win_sync(MPI_Win win)		31
Fortran	2008 binding		32
	sync(win, ierror)		33
	(MPI_Win), INTENT(IN) ::	win	34
	GER, OPTIONAL, INTENT(OUT		35
<b>T</b> (			36
Fortran	0		37
	SYNC(WIN, IERROR)		38
INIE	GER WIN, IERROR		39
The o	call MPI_WIN_SYNC synchron	nizes the private and public window copies of win.	40
		rivate and public window, MPI_WIN_SYNC has the	41
effect of e	nding and reopening an acces	s and exposure epoch on the window (note that it	42

### 12.5.5 Assertions

The assert argument in the calls MPI\_WIN\_POST, MPI\_WIN\_START, MPI\_WIN\_FENCE, MPI\_WIN\_LOCK, and MPI\_WIN\_LOCK\_ALL is used to provide assertions on the context of

does not actually end an epoch or complete any pending MPI RMA operations).

 $44 \\ 45$ 

the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the program—it is erroneous to provide incorrect information. Users may always provide assert = 0 to indicate a general case where no guarantees are made.

Advice to users. Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent shared memory machines. Users should consult their implementation's manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (*End of advice to users.*)

- Advice to implementors. Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (End of advice to implementors.)
- assert is the bit vector OR of zero or more of the following integer constants:

<sup>17</sup> MPI\_MODE\_NOCHECK, MPI\_MODE\_NOSTORE, MPI\_MODE\_NOPUT,

<sup>18</sup> MPI\_MODE\_NOPRECEDE, and MPI\_MODE\_NOSUCCEED. The significant options are listed <sup>19</sup> below for each call.

Advice to users. C/C++ users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (End of advice to users.)

# MPI\_WIN\_START:

MPI\_MODE\_NOCHECK—the matching calls to MPI\_WIN\_POST have already completed on all target processes when the call to MPI\_WIN\_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.

# MPI\_WIN\_POST:

- MPI\_MODE\_NOCHECK—the matching calls to MPI\_WIN\_START have not yet occurred on any origin processes when the call to MPI\_WIN\_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
  - MPI\_MODE\_NOSTORE—the local window was not updated by stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
- MPI\_MODE\_NOPUT—the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

<sup>48</sup> MPI\_WIN\_FENCE:

5

6

7

8

9

10

11 12

13

14

15 16

21

22

23

 $^{24}$ 

25 26

27

28

29

30

31

32

33

34 35

36

37

38

39

40 41

42

43

44

45

46

- MPI\_MODE\_NOSTORE—the local window was not updated by stores (or local get or receive calls) since last synchronization.
- MPI\_MODE\_NOPUT—the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
- MPI\_MODE\_NOPRECEDE—the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
- MPI\_MODE\_NOSUCCEED—the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

#### MPI\_WIN\_LOCK, MPI\_WIN\_LOCK\_ALL:

MPI\_MODE\_NOCHECK—no other process holds, or will attempt to acquire, a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

Advice to users. Note that the nostore and noprecede flags provide information on what happened *before* the call; the noput and nosucceed flags provide information on what will happen *after* the call. (*End of advice to users.*)

#### 12.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the datatype argument of a MPI\_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

As in message-passing, datatypes must be committed before they can be used in RMA communication.

#### 12.6 Error Handling

#### 12.6.1 Error Handlers

Errors occurring during calls to routines that create MPI windows (e.g., MPI\_WIN\_CREATE (...,comm,...)) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked.

The error handler MPI\_ERRORS\_ARE\_FATAL is associated with win during its creation. Users may change this default by explicitly associating a new error handler with win (see Section 9.3).

#### 12.6.2 Error Classes

The error classes for one-sided communication are defined in Table 12.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI\_ERR\_OP or MPI\_ERR\_RANK.

 $\overline{7}$ 

 $^{24}$ 

1	MPI_ERR_WIN	invalid win argument
2	 MPI_ERR_BASE	invalid base argument
3	MPI_ERR_SIZE	invalid size argument
4	MPI_ERR_DISP	invalid disp argument
5	MPI_ERR_LOCKTYPE	invalid locktype argument
6	MPI_ERR_ASSERT	invalid assert argument
7	MPI_ERR_RMA_CONFLICT	conflicting accesses to window
8	MPI_ERR_RMA_SYNC	invalid synchronization of RMA calls
9	MPI_ERR_RMA_RANGE	target memory is not part of the window (in the case
10		of a window created with
11		MPI_WIN_CREATE_DYNAMIC, target memory is not
12		attached)
13	MPI_ERR_RMA_ATTACH	memory cannot be attached (e.g., because of resource
14		exhaustion)
15	MPI_ERR_RMA_SHARED	memory cannot be shared (e.g., some process in the
16		group of the specified communicator cannot expose
17		shared memory)
18	MPI_ERR_RMA_FLAVOR	passed window has the wrong flavor for the called
19		function
20		

Table 12.2: Error classes in one-sided communication routines

### 12.7 Semantics and Correctness

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.

- An RMA operation is completed at the origin by the ensuing call to MPI\_WIN\_COMPLETE, MPI\_WIN\_FENCE, MPI\_WIN\_FLUSH, MPI\_WIN\_FLUSH\_ALL, MPI\_WIN\_FLUSH\_LOCAL, MPI\_WIN\_FLUSH\_LOCAL\_ALL, MPI\_WIN\_UNLOCK, or MPI\_WIN\_UNLOCK\_ALL that synchronizes this access at the origin.
- 2. If an RMA operation is completed at the origin by a call to MPI\_WIN\_FENCE then the operation is completed at the target by the matching call to MPI\_WIN\_FENCE by the target process.
  - 3. If an RMA operation is completed at the origin by a call to MPI\_WIN\_COMPLETE then the operation is completed at the target by the matching call to MPI\_WIN\_WAIT by the target process.
- 4. If an RMA operation is completed at the origin by a call to MPI\_WIN\_UNLOCK,
   MPI\_WIN\_UNLOCK\_ALL, MPI\_WIN\_FLUSH(rank=target), or
   MPI\_WIN\_FLUSH\_ALL, then the operation is completed at the target by that same

call.

- 5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI\_WIN\_POST, MPI\_WIN\_FENCE, MPI\_WIN\_UNLOCK, MPI\_WIN\_UNLOCK\_ALL, or MPI\_WIN\_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.
- 6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI\_WIN\_WAIT, MPI\_WIN\_FENCE, MPI\_WIN\_LOCK, MPI\_WIN\_LOCK\_ALL, or MPI\_WIN\_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy eventually becomes visible in the private copy in process memory without additional RMA calls.

The MPI\_WIN\_FENCE or MPI\_WIN\_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL. In the RMA separate memory model, the update of a private copy in the process memory may be delayed until the target process 20executes a synchronization call on that window (6). Thus, updates to process memory can 21always be delayed in the RMA separate memory model until the process executes a suitable 22synchronization call, while they must complete in the RMA unified model without additional 23synchronization calls. If fence or post-start-complete-wait synchronization is used, updates to a public window copy can be delayed in both memory models until the window owner executes a synchronization call. When passive target synchronization is used, it is necessary to update the public window copy even if the window owner does not execute any related 27synchronization call.

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI\_WIN\_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI\_WIN\_FENCE(0, win2) makes these updates visible in the public copy of win2.

34The behavior of some MPI RMA operations may be *undefined* in certain situations. For 35 example, the result of several origin processes performing concurrent MPI\_PUT operations 36 to the same target location is undefined. In addition, the result of a single origin process 37 performing multiple MPI\_PUT operations to the same target location within the same 38 access epoch is also undefined. The result at the target may have all of the data from one of the MPI\_PUT operations (the "last" one, in some sense), bytes from some of each of the operations, or something else. In MPI-2, such operations were *erroneous*. That meant that an MPI implementation was permitted to raise an error. Thus, user programs or tools that used MPI RMA could not portably permit such operations, even if the application code could function correctly with such an undefined result. Starting with MPI-3, these operations are not erroneous, but do not have a defined behavior.

46Rationale. As discussed in [7], requiring operations such as overlapping puts to 47be erroneous makes it difficult to use MPI RMA to implement programming models— 48 such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further,

1  $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8

9

10

11

12

13 14

15

16

17

18

19

 $^{24}$ 

25

26

28

29

30

31

32

33

39

40

41

42

43

44

while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any implementation that enforces this rule, as it would require significant overhead. Thus, relaxing this condition does not impact existing implementations or applications. (End of rationale.)

Advice to implementors. Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that starting with MPI-3, such operations must not raise an error. (End of advice to implementors.)

A program with a well-defined outcome in the MPI\_WIN\_SEPARATE memory model must obey the following rules.

- S1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.
- S2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates with the same predefined datatype, on the same window. Additional restrictions on the operation apply, see the info key accumulate\_ops in Section 12.2.1.
- $^{24}$ S3. A put or accumulate must not access a target window once a store or a put or accumulate update to another (overlapping) target window has started on a location in 26the target window, until the update becomes visible in the public copy of the win-27dow. Conversely, a store to process memory to a location in a window must not start 28once a put or accumulate update to that target window has started, until the put or 29 accumulate update becomes visible in process memory. In both cases, the restriction 30 applies to operations even if they access disjoint locations in the window.
- Rationale. The last constraint on correct RMA accesses may seem unduly restric-32 tive, as it forbids concurrent accesses to nonoverlapping locations in a window. The 33 34 reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may 35 be needed for locations that were updated by stores and for locations that were re-36 motely updated by put or accumulate operations. Without this constraint, the MPI 37 library would have to track precisely which locations in a window were updated by a 38 put or accumulate call. The additional overhead of maintaining such information is 39 considered prohibitive. (*End of rationale.*) 40
- 41

48

Note that MPI\_WIN\_SYNC may be used within a passive target epoch to synchronize 42the private and public window copies (that is, updates to one are made visible to the other). 43 In the MPI\_WIN\_UNIFIED memory model, the rules are simpler because the public and 44private windows are the same. However, there are restrictions to avoid concurrent access 45to the same memory locations by different processes. The rules that a program with a 46 well-defined outcome must obey in this case are: 47

1

2

3

4

56

7

8

9 10

11

1213

14

15

1617

18

19

20

21

22

23

25

- U1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.
- U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

Advice to users. Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (*End of advice to users.*)

- U3. Updating a location in the window with a store operation that is also the target of a remote read (but not update) is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Store updates will appear in memory, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory, the data remains until replaced by another update. This permits updates to memory with store operations without requiring an RMA epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and elsewhere in this chapter are followed.
- U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate\_ops in Section 12.2.1.
- U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update completes at the target.

Advice to users. In the unified memory model, in the case where the window is in shared memory, MPI\_WIN\_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this

1 2

3

4

5

6

7

8

9

10

11

12

13

14 15

16

17

18

19

20

21 22

23

 $^{24}$ 

25

26

27

28

29

30

31 32

33

34

35

36

37

38

39

40

41

42

43

 $\frac{44}{45}$ 

46

47

routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. MPI\_WIN\_SYNC should be called by the writer before the non-RMA synchronization operation and by the reader after the non-RMA synchronization, as shown in Example 12.21. (*End of advice to users.*)

- A program that violates these rules has undefined behavior.
  - Advice to users. A user can write correct programs by following the following rules:
  - fence: During each period between fence calls, each window is either updated by put or accumulate calls, or updated by stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.
- **post-start-complete-wait:** A window should not be updated with store operations while posted if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.
  - With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.
- lock: Updates to the window are protected by exclusive locks if they may conflict.
   Nonconflicting accesses (such as read-only accesses or accumulate accesses) are
   protected by shared locks, both for load/store accesses and for RMA accesses.
  - changing window or synchronization mode: One can change synchronization mode, or change the window used to access a location that belongs to two overlapping windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI\_WIN\_FENCE, if RMA accesses to the window are synchronized with fences; after a local call to MPI\_WIN\_WAIT, if the accesses are synchronized with post-start-completewait; after the call at the origin (local or remote) to MPI\_WIN\_UNLOCK or MPI\_WIN\_UNLOCK\_ALL if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete.

The RMA synchronization operations define when updates are guaranteed to become visible in public and private windows. Updates may become visible earlier, but such behavior is implementation dependent. (*End of advice to users.*)

The semantics are illustrated by the following examples:

**Example 12.6** The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The MPI\_WIN\_LOCK

1

2

3

4

5

6

7 8

9 10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

29

30

31

32

33

34

35

36

37

38

39

40 41

42

43

44

45 46

47

and MPI\_WIN\_UNLOCK calls around the store to X in process B are necessary to ensure consistency between the public and private copies of the window.

Process A:	Process B: window location X		
	<pre>MPI_Win_lock(EXCLUSIVE, B) store X /* local update to private copy of B */ MPI_Win_unlock(B) /* now visible in public window copy */</pre>		
MPI_Barrier	MPI_Barrier		
<pre>MPI_Win_lock(EXCLUSIVE, B) MPI_Get(X) /* ok, read from public window */ MPI_Win_unlock(B)</pre>			

**Example 12.7** In the RMA unified model, although the public and private copies of the windows are synchronized, caution must be used when combining load/stores and multiprocess synchronization. Although the following example appears correct, the compiler or hardware may delay the store to X after the barrier, possibly resulting in the MPI\_GET returning an incorrect value of X.

Process B: window location X
store X /* update to private & public copy of B */
(PI_Barrier
from window */
7: 7:

MPI\_BARRIER provides process synchronization, but not memory synchronization. The example could potentially be made safe through the use of compiler- and hardware-specific notations to ensure the store to X occurs before process B enters the MPI\_BARRIER. The use of one-sided synchronization calls, as shown in Example 12.6, also ensures the correct result.

**Example 12.8** The following example demonstrates the reading of a memory location updated by a remote process (Rule 6) in the RMA separate memory model. Although the MPI\_WIN\_UNLOCK on process A and the MPI\_BARRIER ensure that the public copy on process B reflects the updated value of X, the call to MPI\_WIN\_LOCK by process B is necessary to synchronize the private copy with the public copy.

Process A:

Process B:

 $\mathbf{2}$ 

	window location X	
MDT Win look (EXCLUST		
<pre>MPI_Win_lock(EXCLUSIVE, B) MPI_Put(X) /* update to public window */</pre>		
MFI_Fut(x) /* update MPI_Win_unlock(B)	to public window */	
II I_WIII_UIIIOCK(D)		
IPI_Barrier	MPI_Barrier	
	MPI_Win_lock(EXCLUSIVE, B) /* now visible in private copy of B */ load X MPI_Win_unlock(B)	
e of exclusive locks g PI_WIN_LOCK synch: oking for changes in X	aple, the barrier is not critical to the semantic correctness. The uarantees a remote process will not modify the public copy after conizes the private and public copies. A polling implementation on process B would be semantically correct. The barrier is required a performs the put operation before process B performs the load of	
nodel, because the load Process B does not nee MPI_WIN_LOCK as the window, the scheduling and hardware specific n explicit one-sided synch	to Example 12.7, the following example is unsafe even in the unified of X can not be guaranteed to occur after the MPI_BARRIER. While d to explicitly synchronize the public and private copies through MPI_PUT will update both the public and private copies of the of the load could result in old values of X being returned. Compiler otations could ensure the load occurs after the data is updated, or ronization calls can be used to ensure the proper result.	
Process A:	Process B:	
	window location X	
'I_Win_lock_all		
1PI_Put(X) /* update 1PI_Win_flush(B)	to window */	
PI_Barrier	MPI_Barrier load X /* may return an obsolete value */	
MPI_Win_unlock_all		
IPI_WIN_LOCK_ALL of rivate copy. Therefore,	following example further clarifies Rule 5. MPI_WIN_LOCK and lo <i>not</i> update the public copy of a window with changes to the there is no guarantee that process A in the following sequence will dated by the local store by process B before the lock.	

CHAPTER 12. ONE-SIDED COMMUNICATIONS

Pro	cess A:	Process B: window location X							
		store X /*	update	to	private	сору	of	В	*/

would be explicitly synchronized with the private copy.

```
1
                              MPI_Win_lock(SHARED, B)
                                                                                        \mathbf{2}
MPI_Barrier
                              MPI_Barrier
                                                                                        3
                                                                                        4
MPI_Win_lock(SHARED, B)
                                                                                        5
MPI_Get(X) /* X may be the X before the store */
                                                                                        6
MPI_Win_unlock(B)
                                                                                        7
                              MPI_Win_unlock(B)
                                                                                        8
                              /* update on X now visible in public window */
                                                                                        9
                                                                                        10
The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would
                                                                                        11
guarantee process A would see the updated value of X, as the public copy of the window
```

**Example 12.11** Similar to the previous example, Rule 5 can have unexpected implications for general active target synchronization with the RMA separate memory model. It is *not* guaranteed that process B reads the value of X as per the local update by process A, because neither MPI\_WIN\_WAIT nor MPI\_WIN\_COMPLETE calls by process A ensure visibility in the public window copy.

```
Process B:
Process A:
window location X
window location Y
store Y
MPI_Win_post(A, B) /* Y visible in public window */
                           MPI_Win_start(A)
MPI_Win_start(A)
store X /* update to private window */
MPI_Win_complete
                           MPI_Win_complete
MPI_Win_wait
/* update on X may not yet visible in public window */
MPI_Barrier
                           MPI_Barrier
                           MPI_Win_lock(EXCLUSIVE, A)
                           MPI_Get(X) /* may return an obsolete value */
                           MPI_Get(Y)
                           MPI_Win_unlock(A)
```

To allow process B to read the value of X stored by A the local store must be replaced by a local MPI\_PUT that updates the public window copy. Note that by this replacement X may become visible in the private copy of process A only after the MPI\_WIN\_WAIT call in process A. The update to Y made before the MPI\_WIN\_POST call is visible in the public window after the MPI\_WIN\_POST call and therefore process B will read the proper value of Y. The MPI\_GET(Y) call could be moved to the epoch started by the MPI\_WIN\_START operation, and process B would still get the value stored by process A.

12

13 14

15

16

17

18

19 20

21

22

23 24

25

26

27 28

29 30

31

32

33 34

35 36

37

38

39

40

41

42

43

44

45

46

 $\mathbf{2}$ 

5

 $\overline{7}$ 

8 9

18

21

2627

28

**Example 12.12** The following example demonstrates the interaction of general active target synchronization with local read operations with the RMA separate memory model. 3 Rules 5 and 6 do not guarantee that the private copy of X at process B has been updated 4 before the load takes place. 6 Process A: Process B: window location X MPI\_Win\_lock(EXCLUSIVE, B) 10 MPI\_Put(X) /\* update to public window \*/ 11MPI\_Win\_unlock(B) 1213MPI\_Barrier MPI\_Barrier 1415MPI\_Win\_post(B) 16MPI\_Win\_start(B) 17 load X /\* access to private window \*/ 19 /\* may return an obsolete value \*/ 20MPI\_Win\_complete 22MPI\_Win\_wait 23 $^{24}$ To ensure that the value put by process A is read, the local load must be replaced with a 25

local MPI\_GET operation, or must be placed after the call to MPI\_WIN\_WAIT.

#### 12.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same 29predefined datatype is as if the accumulates were done at that location in some serial 30 order. Additional restrictions on the operation apply; see the info key accumulate\_ops in  $^{31}$ 32 Section 12.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is 33 34executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to an accumulate operation cannot be 35 accessed by a load or an RMA call other than accumulate until the accumulate operation has 36 completed (at the target). Different interleavings can lead to different results only to the 37 extent that computer arithmetics are not truly associative or commutative. The outcome 38 39 of accumulate operations with overlapping types of different sizes or target displacements is undefined. 40

41

#### 4212.7.2 Ordering

43 Accumulate calls enable element-wise atomic read and write to remote memory locations. 44MPI specifies ordering between accumulate operations from an origin process to the same 45(or overlapping) memory locations at a target process on a per-datatype granularity. The 46 default ordering is strict ordering, which guarantees that overlapping updates from the 47same origin to a remote location are committed in program order and that reads (e.g., with 48

MPI\_GET\_ACCUMULATE) and writes (e.g., with MPI\_ACCUMULATE) are executed and <sup>1</sup> committed in program order. Ordering only applies to operations originating at the same <sup>2</sup> origin that access overlapping target memory regions. MPI does not provide any guarantees <sup>3</sup> for accesses or updates from different origin processes to overlapping target memory regions. <sup>4</sup>

 $\mathbf{5}$ The default strict ordering may incur a significant performance penalty. MPI specifies the info key "accumulate\_ordering" to allow relaxation of the ordering semantics when specified  $\mathbf{6}$  $\overline{7}$ to any window creation function. The values for this key are as follows. If set to "none", then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA in 8 9 MPI-2 but has not been the default since MPI-3. The key can be set to a comma-separated list of required access orderings at the target. Allowed values in the comma-separated list 1011are "rar", "war", "raw", and "waw" for read-after-read, write-after-read, read-after-write, and 12write-after-write ordering, respectively. These indicate whether operations of the specified 13type complete in the order they were issued. For example, "raw" means that any writes must complete at the target before subsequent reads. These ordering requirements apply only to 1415operations issued by the same origin process and targeting the same target process. The 16default value for "accumulate\_ordering" is rar, raw, war, waw, which implies that writes complete 17 at the target in the order in which they were issued, reads complete at the target before any 18 writes that are issued after the reads, and writes complete at the target before any reads 19that are issued after the writes. Any subset of these four orderings can be specified. For 20example, if only read-after-read and write-after-write ordering is required, then the value of 21the "accumulate\_ordering" key could be set to rar, waw. The order of values is not significant.

Note that the above ordering semantics apply only to accumulate operations, not put and get. Put and get within an epoch are unordered.

#### 12.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with point-to-point communication. Access to a target window becomes enabled once the corresponding synchronization (such as MPI\_WIN\_FENCE or MPI\_WIN\_POST) has executed. On the origin process, an RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the communication must complete.

Consider the code fragment in Example 12.4. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 12.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 12.6. Each process updates the window of the other process using a put operation, then accesses its own window. The post calls are nonblocking, and should complete. Once the post calls occur, RMA access to the windows is enabled, so that each process should complete the sequence of calls start-put-complete. Once 48

22 23

24 25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

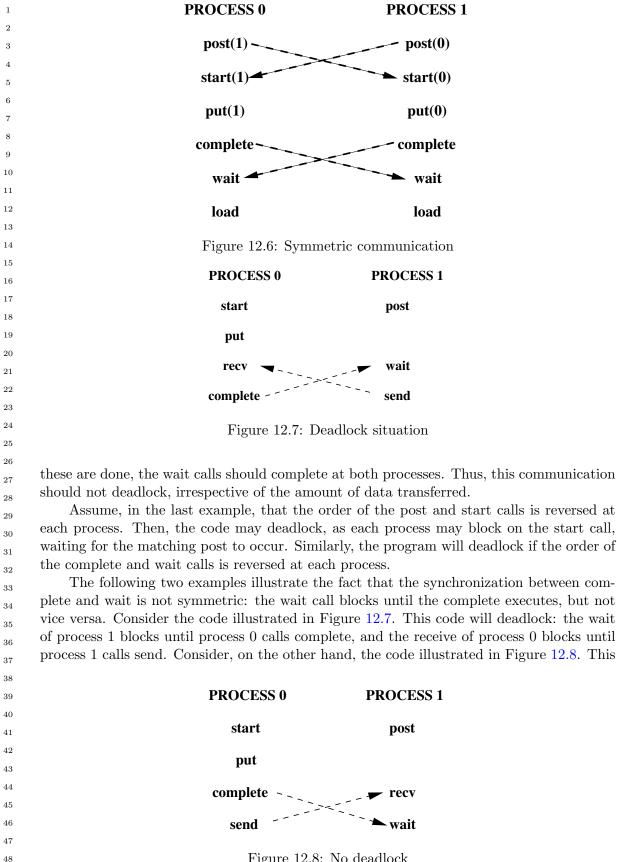


Figure 12.8: No deadlock

code will not deadlock. Once process 1 calls post, then the sequence start, put, complete on process 0 can proceed to completion. Process 0 will reach the send call, allowing the receive call of process 1 to complete.

*Rationale.* MPI implementations must guarantee that a process makes *progress* on all enabled communications it participates in, while blocked on an MPI call. This is true for send-receive communication and applies to RMA communication as well. Thus, in the example in Figure 12.8, the put and complete calls of process 0 should complete while process 1 is blocked on the receive call. This may require the involvement of process 1, e.g., to transfer the data put, while it is blocked on the receive call.

11 A similar issue is whether such progress must occur while a process is busy comput-12ing, or blocked in a non-MPI call. Suppose that in the last example the send-receive 13 pair is replaced by a write-to-socket/read-from-socket pair. Then MPI does not specify whether deadlock is avoided. Suppose that the blocking receive of process 1 is 1415replaced by a very long compute loop. Then, according to one interpretation of the 16MPI standard, process 0 must return from the complete call after a bounded delay, 17 even if process 1 does not reach any MPI call in this period of time. According to 18another interpretation, the complete call may block until process 1 reaches the wait 19call, or reaches another MPI call. The qualitative behavior is the same, under both 20interpretations, unless a process is caught in an infinite compute loop, in which case 21the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this am-2223biguity is unfortunate, the MPI Forum decided not to define which interpretation of  $^{24}$ the standard is the correct one, since the issue is contentious. (End of rationale.)

#### 12.7.4 Registers and Compiler Optimizations

Advice to users. All the material in this section is an advice to users. (End of advice to users.)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI\_WIN\_UNIFIED.

The problem is illustrated by the following code:

Source of Process 1	Source of Process 2	Executed in Process 2	38
bbbb = 777	buff = 999	reg_A:=999	39
call MPI_WIN_FENCE	call MPI_WIN_FENCE		40 41
call MPI_PUT(bbbb		stop appl.thread	41 42
into buff of process 2)		buff:=777 in PUT handler	42
		continue appl.thread	44
call MPI_WIN_FENCE	call MPI_WIN_FENCE		45
	ccc = buff	ccc:=reg_A	46

In this example, variable **buff** is allocated in the register **reg\_A** and therefore **ccc** will have the old value of **buff** and not the new value 777.

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

25 26

27

28

29 30

 $^{31}$ 

32

33

34

35

36

37

47

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 19.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in modules or COMMON blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. Sections 19.1.17 to 19.1.17 discuss several solutions for the problem in this example.

### 12.8 Examples

**Example 12.13** The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

The same code could be written with get rather than put. Note that, during the communication phase, each window is concurrently read (as origin buffer of puts) and written (as target buffer of puts). This is OK, provided that there is no overlap between the target buffer of a put and another communication buffer.

**Example 12.14** Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

```
38
     . . .
39
     while (!converged(A)) {
40
       update_boundary(A);
41
       MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
42
       for(i=0; i < fromneighbors; i++)</pre>
          MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
43
44
                           fromdisp[i], 1, fromtype[i], win);
45
       update_core(A);
46
       MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
47
     }
48
```

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $\overline{7}$ 

The get communication can be concurrent with the core update, since they do not access the same locations, and the local update of the origin buffer by the get call can be concurrent with the local update of the core by the update\_core call. In order to get similar overlap with put communication we would need to use separate windows for the core and for the boundary. This is required because we do not allow local stores to be concurrent with puts on the same, or on overlapping, windows.

Example 12.15 Same code as in Example 12.13, rewritten using post-start-complete-wait. ... while (!converged(A)) { update(A); MPI\_Win\_post(fromgroup, 0, win); MPI\_Win\_start(togroup, 0, win); for(i=0; i < toneighbors; i++) MPI\_Put(&frombuf[i], 1, fromtype[i], toneighbor[i], todisp[i], 1, totype[i], win); MPI\_Win\_complete(win); MPI\_Win\_wait(win); }

**Example 12.16** Same example, with split phases, as in Example 12.14.

**Example 12.17** A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array A0 is updated using values of array A1, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window wini consists of array Ai.

```
...
if (!converged(A0,A1))
    MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Barrier(comm0);
/* the barrier is needed because the start call inside the
loop uses the nocheck option */
```

```
1
     while (!converged(A0, A1)) {
\mathbf{2}
        /* communication on AO and computation on A1 */
3
        update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
4
        MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
5
        for(i=0; i < fromneighbors; i++)</pre>
6
          MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
7
                      fromdisp0[i], 1, fromtype0[i], win0);
8
        update1(A1); /* local update of A1 that is
9
                          concurrent with communication that updates A0 */
10
        MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
11
        MPI_Win_complete(win0);
12
        MPI_Win_wait(win0);
13
14
        /* communication on A1 and computation on A0 */
15
        update2(A0, A1); /* local update of A0 that depends on A1 (and A0) */
16
        MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win1);
17
        for(i=0; i < fromneighbors; i++)</pre>
18
          MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
19
                       fromdisp1[i], 1, fromtype1[i], win1);
20
        update1(A0); /* local update of A0 that depends on A0 only,
21
                        concurrent with communication that updates A1 */
22
        if (!converged(A0,A1))
23
          MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
24
        MPI_Win_complete(win1);
25
        MPI_Win_wait(win1);
26
     }
27
     A process posts the local window associated with win0 before it completes RMA accesses
28
      to the remote windows associated with win1. When the wait(win1) call returns, then all
29
      neighbors of the calling process have posted the windows associated with win0. Conversely,
30
      when the wait(win0) call returns, then all neighbors of the calling process have posted the
^{31}
      windows associated with win1. Therefore, the nocheck option can be used with the calls to
32
      MPI_WIN_START.
33
34
     Put calls can be used, instead of get calls, if the area of array A0 (resp. A1) used by the
     update(A1, A0) (resp. update(A0, A1)) call is disjoint from the area modified by the RMA
35
     communication. On some systems, a put call may be more efficient than a get call, as it
36
     requires information exchange only in one direction.
37
38
         In the next several examples, for conciseness, the expression
39
40
     z = MPI_Get_accumulate(...)
41
     means to perform an MPI_GET_ACCUMULATE with the result buffer (given by result_addr
42
     in the description of MPI_GET_ACCUMULATE) on the left side of the assignment, in this
43
     case, z. This format is also used with MPI_COMPARE_AND_SWAP and MPI_COMM_SIZE.
44
     Process B... refers to any process other than A.
45
46
     Example 12.18 The following example implements a naive, non-scalable counting sema-
47
     phore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy
```

of X, as well as MPI\_WIN\_FLUSH to complete operations without ending the access epoch opened with MPI\_WIN\_LOCK\_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI\_ACCUMULATE and MPI\_GET\_ACCUMULATE are used to write to or read from the local public copy.

Process A:	Process B:
MPI_Win_lock_all	MPI_Win_lock_all
window location X	
X=MPI_Comm_size()	
MPI_Win_sync	
MPI_Barrier	MPI_Barrier
<pre>MPI_Accumulate(X, MPI_SUM, -1)</pre>	MPI_Accumulate(X, MPI_SUM, -1)
stack variable z	stack variable z
do	do
<pre>z = MPI_Get_accumulate(X,</pre>	<pre>z = MPI_Get_accumulate(X,</pre>
MPI_NO_OP, 0)	MPI_NO_OP, O)
MPI_Win_flush(A)	MPI_Win_flush(A)
while(z!=0)	while(z!=0)

**Example 12.19** Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI\_WIN\_LOCK\_ALL and MPI\_WIN\_UNLOCK\_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

Process A: window location X window location T	Process B: window location Y
<pre>MPI_Win_lock_all X=1 MPI_Win_sync MPI_Barrier MPI_Accumulate(T, MPI_REPLACE, 1) stack variables t,y t=1 y=MPI_Get_accumulate(Y,     MPI_N0_OP, 0) while(y==1 &amp;&amp; t==1) do     y=MPI_Get_accumulate(Y,         MPI_N0_OP, 0)</pre>	<pre>MPI_Win_lock_all Y=1 MPI_Win_sync MPI_Barrier MPI_Accumulate(T, MPI_REPLACE, 0) stack variable t,x t=0 x=MPI_Get_accumulate(X,     MPI_N0_0P, 0) while(x==1 &amp;&amp; t==0) do     x=MPI_Get_accumulate(X,         MPI_N0_0P, 0)</pre>
t=MPI_Get_accumulate(T,	t=MPI_Get_accumulate(T,

 $\mathbf{2}$ 

MPI_NO_OP, 0) MPI_Win_flush_all	MPI_NO_OP, 0) MPI_Win_flush(A)
done	done
// critical region	<pre>// critical region</pre>
MPI_Accumulate(X, MPI_REPLACE, 0)	MPI_Accumulate(Y, MPI_REPLACE, 0)
MPI_Win_unlock_all	MPI_Win_unlock_all

**Example 12.20** Implementing a critical region between multiple processes with compare and swap. The call to MPI\_WIN\_SYNC is necessary on Process A after local initialization of A to guarantee the public copy has been updated with the initialization value found in the private copy. It would also be valid to call MPI\_ACCUMULATE with MPI\_REPLACE to directly initialize the public copy. A call to MPI\_WIN\_FLUSH would be necessary to assure A in the public copy of Process A had been updated before the barrier.

Process A:	Process B:
MPI_Win_lock_all	MPI_Win_lock_all
atomic location A	
A=O	
MPI_Win_sync	
MPI_Barrier	MPI_Barrier
stack variable r=1	stack variable r=1
while(r != 0) do	while(r != 0) do
<pre>r = MPI_Compare_and_swap(A, 0, 1)</pre>	r = MPI_Compare_and_swap(A, 0, 1)
MPI_Win_flush(A)	MPI_Win_flush(A)
done	done
// critical region	// critical region
<pre>r = MPI_Compare_and_swap(A, 1, 0)</pre>	r = MPI_Compare_and_swap(A, 1, 0)
MPI_Win_unlock_all	MPI_Win_unlock_all

**Example 12.21** The following example demonstrates the proper synchronization in the unified memory model when a data transfer is implemented with load and store in the case of windows in shared memory (instead of MPI\_PUT or MPI\_GET) and the synchronization between processes is performed using point-to-point communication. The synchronization between processes must be supplemented with a memory synchronization through calls to MPI\_WIN\_SYNC, which act locally as a processor-memory barrier. In Fortran, if MPI\_ASYNC\_PROTECTS\_NONBLOCKING is .FALSE. or the variable X is not declared as ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with MPI\_F\_SYNC\_REG operations. (No equivalent function is needed in C.) The variable X is contained within a shared memory window and X corresponds to the same memory location at both processes. The MPI\_WIN\_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI\_WIN\_SYNC operation performed by process B ensures that process A's updates to X are visible to process В. Process A: Process B: MPI\_WIN\_LOCK\_ALL( MPI\_WIN\_LOCK\_ALL(

MPI\_MODE\_NOCHECK, win)

 $^{31}$ 

MPI\_MODE\_NOCHECK, win)

-		
		1
DO	DO	2
X=		3
		4
MPI_F_SYNC_REG(X)		5
MPI_WIN_SYNC(win)		6
MPI_SEND	MPI_RECV	7
	MPI_WIN_SYNC(win)	8
	MPI_F_SYNC_REG(X)	9
		10
	print X	11
		12
	MPI_F_SYNC_REG(X)	13
MPI_RECV	MPI_SEND	14
MPI_F_SYNC_REG(X)		15
END DO	END DO	16
		17
MPI_WIN_UNLOCK_ALL(win)	MPI_WIN_UNLOCK_ALL(win)	18
		19

**Example 12.22** The following example shows how request-based operations can be used to overlap communication with computation. Each process fetches, processes, and writes the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to allow up to M communication operations to overlap with computation.

```
int
            i, j;
MPI_Win
            win;
MPI_Request put_req[M] = { MPI_REQUEST_NULL };
MPI_Request get_req;
double
            *baseptr;
double
            data[M][N];
MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
 MPI_COMM_WORLD, &baseptr, &win);
MPI_Win_lock_all(0, win);
for (i = 0; i < NSTEPS; i++) {</pre>
 if (i<M)
   j=i;
 else
   MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);
 MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
          &get_req);
 MPI_Wait(&get_req,MPI_STATUS_IGNORE);
 compute(i, data[j], ...);
 MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
```

627

20

21

22

23

 $\frac{24}{25}$ 

26

27

28

29

30 31

32 33

34

35 36

37

38 39

40 41

42

43

44

 $45 \\ 46$ 

 $\mathbf{2}$ 

3 4

5

6 7

8

9

10

11

12

13

14 15

16

17 18

19

20

21

22 23

24

25

26

27

28 29

30

31

32

33

34 35

36 37

38

39

40

41 42

43

44

45

46 47

```
&put_req[j]);
}
MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
MPI_Win_unlock_all(win);
Example 12.23 The following example constructs a distributed shared linked list using
dynamic windows. Initially process 0 creates the head of the list, attaches it to the window,
and broadcasts the pointer to all processes. All processes then concurrently append N new
elements to the list. When a process attempts to attach its element to the tail of the
list it may discover that its tail pointer is stale and it must chase ahead to the new tail
before the element can be attached. This example requires some modification to work in an
environment where the layout of the structures is different on different processes.
. . .
#define NUM_ELEMS 10
#define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
                                  offsetof(llist_ptr_t, rank) )
#define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) + \
                                 offsetof(llist_ptr_t, disp) )
/* Linked list pointer */
typedef struct {
  MPI_Aint disp;
            rank;
  int
} llist_ptr_t;
/* Linked list element */
typedef struct {
  llist_ptr_t next;
  int value;
} llist_elem_t;
const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };
/* List of locally allocated list elements. */
static llist_elem_t **my_elems = NULL;
static int my_elems_size = 0;
static int my_elems_count = 0;
/* Allocate a new shared linked list element */
MPI_Aint alloc_elem(int value, MPI_Win win) {
  MPI_Aint disp;
  llist_elem_t *elem_ptr;
  /* Allocate the new element and register it with the window */
```

```
MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
  elem_ptr->value = value;
  elem_ptr->next = nil;
  MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));
  /* Add the element to the list of local elements so we can free
     it later. */
  if (my_elems_size == my_elems_count) {
   my_elems_size += 100;
   my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
  }
  my_elems[my_elems_count] = elem_ptr;
  my_elems_count++;
  MPI_Get_address(elem_ptr, &disp);
  return disp;
}
int main(int argc, char *argv[]) {
                procid, nproc, i;
  int
  MPI_Win
                llist_win;
  llist_ptr_t
                head_ptr, tail_ptr;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &procid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);
  /* Process 0 creates the head node */
  if (procid == 0)
    head_ptr.disp = alloc_elem(-1, llist_win);
  /* Broadcast the head pointer to everyone */
  head_ptr.rank = 0;
  MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
  tail_ptr = head_ptr;
  /* Lock the window for shared access to all targets */
  MPI_Win_lock_all(0, llist_win);
  /* All processes concurrently append NUM_ELEMS elements to the list */
  for (i = 0; i < NUM_ELEMS; i++) {</pre>
    llist_ptr_t new_elem_ptr;
    int success;
```

1

 $\mathbf{2}$ 

3

4

5

7

8

9

10

11

12

13

14 15

16

17

18 19

20

21

22

23 24

25 26

27

28 29

30 31

32

33

34 35

36

37

38

39 40

41

42 43

44

45

46

```
1
         /* Create a new list element and attach it to the window */
2
         new_elem_ptr.rank = procid;
3
         new_elem_ptr.disp = alloc_elem(procid, llist_win);
4
5
         /* Append the new node to the list. This might take multiple
6
            attempts if others have already appended and our tail pointer
7
             is stale. */
8
         do {
9
           llist_ptr_t next_tail_ptr = nil;
10
11
           MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
12
                (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
13
               MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
14
                llist_win);
15
16
           MPI_Win_flush(tail_ptr.rank, llist_win);
17
           success = (next_tail_ptr.rank == nil.rank);
18
19
           if (success) {
20
             MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
21
                  MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
22
                  MPI_AINT, MPI_REPLACE, llist_win);
23
24
             MPI_Win_flush(tail_ptr.rank, llist_win);
25
              tail_ptr = new_elem_ptr;
26
27
           } else {
28
              /* Tail pointer is stale, fetch the displacement. May take
29
                 multiple tries if it is being updated. */
30
             do {
31
               MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
32
                    1, MPI_AINT, tail_ptr.rank,
33
                    MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
34
                    1, MPI_AINT, MPI_NO_OP, llist_win);
35
36
               MPI_Win_flush(tail_ptr.rank, llist_win);
37
              } while (next_tail_ptr.disp == nil.disp);
38
              tail_ptr = next_tail_ptr;
39
           }
40
         } while (!success);
41
       }
42
43
       MPI_Win_unlock_all(llist_win);
44
       MPI_Barrier(MPI_COMM_WORLD);
45
46
       /* Free all the elements in the list */
47
       for ( ; my_elems_count > 0; my_elems_count--) {
48
```

```
MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
MPI_Free_mem(my_elems[my_elems_count-1]);
}
MPI_Win_free(&llist_win);
```

 $\mathbf{2}$ 

 $\mathbf{5}$ 

### Chapter 13

## **External Interfaces**

#### 13.1 Introduction

This chapter contains calls used to create **generalized requests**, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. These calls can be used to layer new functionality on top of MPI. Section 13.3 deals with setting the information found in **status**. This functionality is needed for generalized requests.

 $^{24}$ 

#### 13.2 Generalized Requests

The goal of generalized requests is to allow users to define new nonblocking operations. Such an outstanding nonblocking operation is represented by a (generalized) request. A fundamental property of nonblocking operations is that *progress* toward the completion of this operation occurs asynchronously, i.e., concurrently with normal program execution. Typically, this requires execution of code concurrently with the execution of the user code, e.g., in a separate thread or in a signal handler. Operating systems provide a variety of mechanisms in support of concurrent execution. MPI does not attempt to standardize or to replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI\_WAIT or MPI\_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

*Rationale.* It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI through a call to

```
1
     MPI_GREQUEST_COMPLETE when the operation completes. MPI maintains the "comple-
\mathbf{2}
     tion" status of generalized requests. Any other request state has to be maintained by the
3
     user.
4
          A new generalized request is started with
5
6
     MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
7
8
       IN
                                              callback function invoked when request status is
                 query_fn
9
                                              queried (function)
10
       IN
                 free_fn
                                              callback function invoked when request is freed
11
                                              (function)
12
                 cancel_fn
       IN
                                              callback function invoked when request is cancelled
13
                                              (function)
14
15
       IN
                 extra_state
                                              extra state
16
       OUT
                 request
                                              generalized request (handle)
17
18
     C binding
19
     int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
20
                     MPI_Grequest_free_function *free_fn,
21
                     MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
22
                     MPI_Request *request)
23
^{24}
     Fortran 2008 binding
25
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
26
                     ierror)
27
          PROCEDURE(MPI_Grequest_query_function) :: query_fn
28
          PROCEDURE(MPI_Grequest_free_function) :: free_fn
29
          PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
30
          INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
31
          TYPE(MPI_Request), INTENT(OUT) :: request
32
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     Fortran binding
34
     MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
35
                     IERROR)
36
          EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
37
          INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
38
          INTEGER REQUEST, IERROR
39
40
41
           Advice to users.
                              Note that a generalized request is of the same type as regular
42
           requests, in C and Fortran. (End of advice to users.)
43
         The call starts a generalized request and returns a handle to it in request.
44
         The syntax and meaning of the callback functions are listed below. All callback func-
45
     tions are passed the extra_state argument that was associated with the request by the
46
47
     starting call MPI_GREQUEST_START; extra_state can be used to maintain user-defined
48
     state for the request.
```

CHAPTER 13. EXTERNAL INTERFACES

request is freed.

```
1
    In C, the query function is
                                                                                           \mathbf{2}
typedef int MPI_Grequest_query_function(void *extra_state,
                                                                                           3
               MPI_Status *status);
                                                                                           4
in Fortran with the mpi_f08 module
                                                                                           5
ABSTRACT INTERFACE
                                                                                           6
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
    TYPE(MPI_Status) :: status
                                                                                           9
    INTEGER :: ierror
                                                                                          10
                                                                                          11
in Fortran with the mpi module and mpif.h
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
                                                                                          12
                                                                                          13
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                          14
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
                                                                                          15
    The query_fn function computes the status that should be returned for the generalized
                                                                                          16
request. The status also includes information about successful/unsuccessful cancellation of
                                                                                          17
the request (result to be returned by MPI_TEST_CANCELLED).
                                                                                          18
    The query_fn callback is invoked by the MPI_{WAIT|TEST}{ANY|SOME|ALL} call that
                                                                                          19
completed the generalized request associated with this callback. The callback function is
                                                                                          20
also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when
                                                                                          21
the call occurs. In both cases, the callback is passed a reference to the corresponding
                                                                                          22
status variable passed by the user to the MPI call; the status set by the callback function
                                                                                          23
is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or
                                                                                          ^{24}
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI
                                                                                          25
will pass a valid status object to query_fn, and this status will be ignored upon return of the
                                                                                          26
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE
                                                                                          27
is called on the request; it may be invoked several times for the same generalized request,
                                                                                          28
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also
                                                                                          29
that a call to MPI_{WAIT|TEST}{SOME|ALL} may cause multiple invocations of query_fn
                                                                                          30
callback functions, one for each generalized request that is completed by the MPI call. The
                                                                                          ^{31}
order of these invocations is not specified by MPI.
                                                                                          32
    In C, the free function is
                                                                                          33
typedef int MPI_Grequest_free_function(void *extra_state);
                                                                                          34
                                                                                          35
in Fortran with the mpi_f08 module
                                                                                          36
ABSTRACT INTERFACE
                                                                                          37
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
                                                                                          38
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                          39
    INTEGER :: ierror
                                                                                          40
in Fortran with the mpi module and mpif.h
                                                                                          41
SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
                                                                                          42
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                          43
    INTEGER IERROR
                                                                                          44
                                                                                          45
The free_fn function is invoked to clean up user-allocated resources when the generalized
                                                                                          46
```

<sup>1</sup> The free\_fn callback is invoked by the MPI\_{WAIT|TEST}{ANY|SOME|ALL} call that <sup>2</sup> completed the generalized request associated with this callback. free\_fn is invoked after <sup>3</sup> the call to query\_fn for the same request. However, if the MPI call completed multiple <sup>4</sup> generalized requests, the order in which free\_fn callback functions are invoked is not specified <sup>5</sup> by MPI.

6 The free\_fn callback is also invoked for generalized requests that are freed by a call 7to MPI\_REQUEST\_FREE (no call to MPI\_{WAIT|TEST}{ANY|SOME|ALL} will occur for 8 such a request). In this case, the callback function will be called either in the MPI call 9 MPI\_REQUEST\_FREE(request), or in the MPI call MPI\_GREQUEST\_COMPLETE(request), 10 whichever happens last, i.e., in this case the actual freeing code is executed as soon as both 11calls MPI\_REQUEST\_FREE and MPI\_GREQUEST\_COMPLETE have occurred. The request 12is not deallocated until after free\_fn completes. Note that free\_fn will be invoked only once 13per request by a correct program.

Advice to users. Calling MPI\_REQUEST\_FREE(request) will cause the request handle 15to be set to MPI\_REQUEST\_NULL. This handle to the generalized request is no longer 16valid. However, user copies of this handle are valid until after free\_fn completes since 17 MPI does not deallocate the object until then. Since free\_fn is not called until after 18 MPI\_GREQUEST\_COMPLETE, the user copy of the handle can be used to make this 19 call. Users should note that MPI will deallocate the object after free\_fn executes. At 20this point, user copies of the request handle no longer point to a valid request. MPI will 21not set user copies to MPI\_REQUEST\_NULL in this case, so it is up to the user to avoid 22accessing this stale handle. This is a special case in which MPI defers deallocating the 23object until a later time that is known by the user. (End of advice to users.)  $^{24}$ 

In C, the cancel function is

```
<sup>28</sup> in Fortran with the mpi_f08 module
```

```
<sup>29</sup> ABSTRACT INTERFACE
```

<sup>30</sup> SUBROUTINE MPI\_Grequest\_cancel\_function(extra\_state, complete, ierror)
 <sup>31</sup> INTEGER(KIND=MPI\_ADDRESS\_KIND) :: extra\_state
 <sup>32</sup> LOGICAL :: complete

<sup>33</sup> INTEGER :: ierror

```
in Fortran with the mpi module and mpif.h
```

```
    <sup>35</sup> IN FOOTMAR WIGH the mp1 module and mp11.1
    <sup>36</sup> SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
    <sup>37</sup> INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

```
38 LOGICAL COMPLETE
```

```
39 INTEGER IERROR
```

The cancel\_fn function is invoked to start the cancelation of a generalized request. It is called by MPI\_CANCEL(request). MPI passes complete = true to the callback function if MPI\_GREQUEST\_COMPLETE was already called on the request, and complete = false otherwise.

All callback functions return an error code. The code is passed back and dealt with as appropriate for the error code by the MPI function that invoked the callback function. For example, if error codes are returned then the error code returned by the callback function will be returned by the MPI function that invoked the callback function. In the case of an MPI\_{WAIT|TEST}{ANY} call that invokes both query\_fn and free\_fn, the MPI call will

14

return the error code returned by the last callback, namely free\_fn. If one or more of the requests in a call to MPI\_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return MPI\_ERR\_IN\_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its free\_fn callback function. However, if the MPI function was passed MPI\_STATUSES\_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query\_fn must not set the error field of status since query\_fn may be called by MPI\_WAIT or MPI\_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query\_fn is invoked and can decide correctly when to put the returned error code in the error field of status. (End of advice to users.)

INOUT	request	generalized request	(handle)
-------	---------	---------------------	----------

#### C binding

int MPI\_Grequest\_complete(MPI\_Request request)

#### Fortran 2008 binding

MPI\_Grequest\_complete(request, ierror)
 TYPE(MPI\_Request), INTENT(IN) :: request
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

#### Fortran binding

MPI\_GREQUEST\_COMPLETE(REQUEST, IERROR) INTEGER REQUEST, IERROR

The call informs MPI that the operations represented by the generalized request request are complete (see definitions in Section 2.4). A call to MPI\_WAIT(request, status) will return and a call to MPI\_TEST(request, flag, status) will return flag = true only after a call to MPI\_GREQUEST\_COMPLETE has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI\_TEST, MPI\_REQUEST\_FREE, or MPI\_CANCEL still hold. For example, these calls are supposed to be local and nonblocking. Therefore, the callback functions query\_fn, free\_fn, or cancel\_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI\_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

Advice to implementors. A call to MPI\_GREQUEST\_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

#### 13.2.1 Examples

3 Example 13.1 This example shows the code for a user-defined reduce operation on an int 4 using a binary tree: each non-root node receives two messages, sums them, and sends them  $\mathbf{5}$ up. We assume that no status is returned and that the operation cannot be cancelled. 6  $\overline{7}$ typedef struct { 8 MPI\_Comm comm; 9 int tag; 10 int root; 11int valin; 12int \*valout; 13MPI\_Request request; 14} ARGS; 151617int myreduce(MPI\_Comm comm, int tag, int root,  $^{18}$ int valin, int \*valout, MPI\_Request \*request) 19{ 20ARGS \*args; 21pthread\_t thread; 22 23/\* start request \*/  $^{24}$ MPI\_Grequest\_start(query\_fn, free\_fn, cancel\_fn, NULL, request); 2526args = (ARGS\*)malloc(sizeof(ARGS)); 27args->comm = comm; 28args->tag = tag; 29args->root = root; 30 args->valin = valin;  $^{31}$ args->valout = valout; 32 args->request = \*request; 33 34/\* spawn thread to handle request \*/ 35 /\* The availability of the pthread\_create call is system dependent \*/ 36 pthread\_create(&thread, NULL, reduce\_thread, args); 37 38 return MPI\_SUCCESS; 39} 4041 /\* thread code \*/ 42void\* reduce\_thread(void \*ptr) 43 { 44int lchild, rchild, parent, lval, rval, val; 45MPI\_Request req[2]; 46ARGS \*args; 4748

1

 $\mathbf{2}$ 

```
1
   args = (ARGS*)ptr;
                                                                                    \mathbf{2}
                                                                                    3
   /* compute left and right child and parent in tree; set
                                                                                    4
      to MPI_PROC_NULL if does not exist */
                                                                                    5
   /* code not shown */
                                                                                    6
   . . .
                                                                                    7
                                                                                    8
   MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
                                                                                    9
   MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
                                                                                    10
   MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
                                                                                    11
   val = lval + args->valin + rval;
                                                                                    12
   MPI_Send(&val, 1, MPI_INT, parent, args->tag, args->comm);
                                                                                    13
   if (parent == MPI_PROC_NULL) *(args->valout) = val;
                                                                                    14
   MPI_Grequest_complete((args->request));
                                                                                    15
   free(ptr);
                                                                                    16
   return(NULL);
                                                                                    17
}
                                                                                    18
                                                                                    19
int query_fn(void *extra_state, MPI_Status *status)
                                                                                    20
{
                                                                                    21
   /* always send just one int */
                                                                                    22
   MPI_Status_set_elements(status, MPI_INT, 1);
                                                                                    23
   /* can never cancel so always true */
                                                                                    24
   MPI_Status_set_cancelled(status, 0);
                                                                                    25
   /* choose not to return a value for this */
                                                                                    26
   status->MPI_SOURCE = MPI_UNDEFINED;
                                                                                    27
   /* tag has no meaning for this generalized request */
                                                                                    28
   status->MPI_TAG = MPI_UNDEFINED;
                                                                                    29
   /* this generalized request never fails */
                                                                                    30
   return MPI_SUCCESS;
                                                                                    31
}
                                                                                    32
                                                                                    33
                                                                                    34
int free_fn(void *extra_state)
                                                                                    35
{
                                                                                    36
   /* this generalized request does not need to do any freeing */
                                                                                    37
   /* as a result it never fails here */
                                                                                    38
   return MPI_SUCCESS;
                                                                                    39
}
                                                                                    40
                                                                                    41
                                                                                    42
int cancel_fn(void *extra_state, int complete)
                                                                                    43
ſ
                                                                                    44
   /* This generalized request does not support cancelling.
                                                                                    45
      Abort if not already done. If done then treat as if cancel failed.*/
                                                                                    46
   if (!complete) {
                                                                                    47
     fprintf(stderr,
                                                                                    48
```

```
1
2
3
4
5
6
```

8

```
"Cannot cancel generalized request - aborting program\n");
MPI_Abort(MPI_COMM_WORLD, 99);
}
return MPI_SUCCESS;
```

### 13.3 Associating Information with Status

<sup>9</sup> MPI supports several different types of requests besides those for point-to-point operations. <sup>10</sup> These range from MPI calls for I/O to generalized requests. It is desirable to allow these <sup>12</sup> calls to use the same request mechanism, which allows one to wait or test on different <sup>13</sup> types of requests. However, MPI\_{TEST|WAIT}{ANY|SOME|ALL} returns a status with <sup>14</sup> information about the request. With the generalization of requests, one needs to define <sup>15</sup> what information will be returned in the status object.

Each MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI\_{TEST|WAIT}{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The fields with meaningful values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in the status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

```
27
28
```

34

40

41

43

MPI\_STATUS\_SET\_ELEMENTS(status, datatype, count)

```
    INOUT status status with which to associate count (status)
    IN datatype datatype associated with count (handle)
    IN count number of elements to associate with status (integer)
```

#### C binding

```
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
int count)
```

```
<sup>38</sup> Fortran 2008 binding
```

```
<sup>39</sup> MPI_Status_set_elements(status, datatype, count, ierror)
```

```
TYPE(MPI_Status), INTENT(INOUT) :: status
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
<sup>42</sup> INTEGER, INTENT(IN) :: count
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
44
45 Fortran binding
```

```
46 MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
```

```
47 INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

48

}

MPI_STAT	TUS_SET_ELEMENTS_X(stat	us, datatype, count)	1		
INOUT	status	status with which to associate count (status)	2		
IN	datatype	datatype associated with count (handle)	3 4		
IN	count	number of elements to associate with status (integer)	5		
	count	number of elements to appointe with status (meeger)	6		
C bindin	C binding				
<pre>int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,</pre>					
	MPI_Count count)		9 10		
Fortran 2	2008 binding		10		
	MPI_Status_set_elements_x(status, datatype, count, ierror)				
TYPE	TYPE(MPI_Status), INTENT(INOUT) :: status				
	TYPE(MPI_Datatype), INTENT(IN) :: datatype				
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count INTEGER, OPTIONAL, INTENT(OUT) :: ierror					
LNIE	JER, UPIIUNAL, INIENI(UUI	) :: lerror	16 17		
	Fortran binding				
	MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)				
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT					
			21		
These functions modify the opaque part of status so that a call to					
MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X will return count. MPI_GET_COUNT will return a compatible value.					
will return	i a compatible value.		24 25		
Rati	onale. The number of eleme	ents is set instead of the count because the former	26		
can	deal with a nonintegral numb	er of datatypes. (End of rationale.)	27		
A	answert call to MDL CET (	OUNT(status, datature, sount)	28		
A subsequent call to MPI_GET_COUNT(status, datatype, count), MPI_GET_ELEMENTS(status, datatype, count), or					
	MPI_GET_ELEMENTS(status, datatype, count) must use a datatype argument that has				
the same type signature as the datatype argument that was used in the call to					
MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X.					
י ת			34		
	<i>Rationale.</i> The requirement of matching type signatures for these calls is similar to the restriction that holds when <b>count</b> is set by a receive operation: in that case,				
	the calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X				
	must use a <b>datatype</b> with the same signature as the datatype used in the receive call				
(Eno	d of rationale.)		38 39		
			40		
			41		
MPI_STATUS_SET_CANCELLED(status, flag)					
INOUT	status	status with which to associate cancel flag (status)	43		
IN	flag	if true, indicates request was cancelled (logical)	44 45		
	-0	,	45		
C bindin	g		47		
	0	Status *status, int flag)	48		

1 2 3 4 5 6 7 8 9	<pre>Fortran 2008 binding MPI_Status_set_cancelled(status, flag, ierror)     TYPE(MPI_Status), INTENT(INOUT) :: status     LOGICAL, INTENT(IN) :: flag     INTEGER, OPTIONAL, INTENT(OUT) :: ierror Fortran binding MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)     INTEGER STATUS(MPI_STATUS_SIZE), IERROR</pre>
10	LOGICAL FLAG
11 12 13	If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will also return flag = true, otherwise it will return false.
13 14 15 16 17 18 19 20 21 20 21 22 23 24 25 26 27 28	Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable results and is strongly discouraged. ( <i>End of advice to users.</i> )
28 29 30	
31	
32 33	
34	
35	
36 37	
38	
39	
40	
41 42	
42 43	
44	
45	
46	
47	
48	

### Chapter 14

# I/O

#### 14.1 Introduction

POSIX provides a model of a widely portable file system, but the portability and optimization needed for parallel I/O cannot be achieved with the POSIX interface.

The significant optimizations required for efficiency (e.g., grouping [54], collective buffering [8, 16, 55, 59, 66], and disk-directed I/O [48]) can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

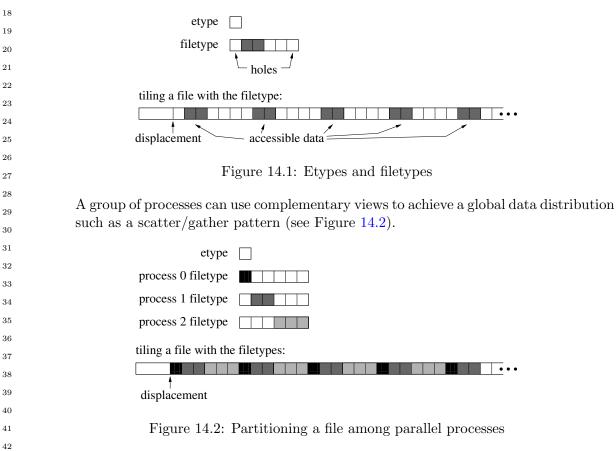
#### 14.1.1 Definitions

- file An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group.
- **displacement** A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement."
- etype An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

 $^{24}$ 

filetype A *filetype* is the basis for partitioning a file among processes and defines a template
 for accessing the file. A filetype is either a single etype or a derived MPI datatype
 constructed from multiple instances of the same etype. In addition, the extent of any
 hole in the filetype must be a multiple of the etype's extent. The displacements in the
 typemap of the filetype are not required to be distinct, but they must be non-negative
 and monotonically nondecreasing.

view A view defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI\_TYPE\_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 14.1 shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI\_BYTE).



offset An offset is a position in the file relative to the current view, expressed as a count of
 etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0
 is the location of the first etype visible in the view (after skipping the displacement and
 any initial holes in the view). For example, an offset of 2 for process 1 in Figure 14.2 is
 the position of the eighth etype in the file after the displacement. An "explicit offset"
 is an offset that is used as an argument in explicit data access routines.

- file size and end of file The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.
- file pointer A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.
- file handle A *file handle* is an opaque object created by MPI\_FILE\_OPEN and freed by MPI\_FILE\_CLOSE. All operations on an open file reference the file through the file handle.

# 14.2 File Manipulation

14.2.1 Opening a File

#### MPI\_FILE\_OPEN(comm, filename, amode, info, fh)

IN	comm	communicator (handle)	21
		× /	22
IN	filename	name of file to open (string)	23
IN	amode	file access mode (integer)	24
IN	info	info object (handle)	25
	fh		26
OUT	fh	new file handle (handle)	27

## C binding

#### Fortran 2008 binding

MPI\_File\_open(comm, filename, amode, info, fh, ierror)
 TYPE(MPI\_Comm), INTENT(IN) :: comm
 CHARACTER(LEN=\*), INTENT(IN) :: filename
 INTEGER, INTENT(IN) :: amode
 TYPE(MPI\_Info), INTENT(IN) :: info
 TYPE(MPI\_File), INTENT(OUT) :: fh
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Fortran binding

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
INTEGER COMM, AMODE, INFO, FH, IERROR
CHARACTER*(*) FILENAME
```

MPI\_FILE\_OPEN opens the file identified by the file name filename on all processes in the comm communicator group. MPI\_FILE\_OPEN is a collective routine: all processes must provide the same value for amode, and all processes must provide filenames that reference

1 the same file. (Values for info may vary.) comm must be an intra-communicator; it is  $\mathbf{2}$ erroneous to pass an inter-communicator to MPI\_FILE\_OPEN. Errors in MPI\_FILE\_OPEN 3 are raised using the default file error handler (see Section 14.7). When using the World 4 Model (Section 11.1), a process can open a file independently of other processes by using  $\mathbf{5}$ the MPI\_COMM\_SELF communicator. Applications using the Sessions Model (Section 11.3) 6 can achieve the same result using communicators created from the "mpi://SELF" process  $\overline{7}$ set. The file handle returned, fh, can be subsequently used to access the file until the file is 8 closed using MPI\_FILE\_CLOSE. Before calling MPI\_FINALIZE, the user is required to close 9 (via MPI\_FILE\_CLOSE) all files that were opened with MPI\_FILE\_OPEN. Note that the 10 communicator comm is unaffected by MPI\_FILE\_OPEN and continues to be usable in all 11MPI routines (e.g., MPI\_SEND). Furthermore, the use of comm will not interfere with I/O 12behavior.

The format for specifying the file name in the filename argument is implementation dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that filename include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of ufs:), a remote hostname (e.g., a prefix of machine.univ.edu:), or a file password (e.g., a suffix of /PASSWORD=SECRET). (End of advice to implementors.)

Advice to users. On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "/tmp/foo" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

27 28

29

30

 $^{31}$ 

32

33

34 35

36

37 38

39

40 41

42 43

44

 $45 \\ 46$ 

47 48

15

16

17

18

19

20 21

22

23

24

25

26

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the MPI\_FILE\_SET\_VIEW routine.

The following access modes are supported (specified in **amode**, a bit vector **OR** of the following integer constants):

- MPI\_MODE\_RDONLY—read only,
- MPI\_MODE\_RDWR—reading and writing,
- MPI\_MODE\_WRONLY—write only,
- MPI\_MODE\_CREATE—create the file if it does not exist,
- MPI\_MODE\_EXCL—error if creating file that already exists,
- MPI\_MODE\_DELETE\_ON\_CLOSE—delete file on close,
- MPI\_MODE\_UNIQUE\_OPEN—file will not be concurrently opened elsewhere,
- MPI\_MODE\_SEQUENTIAL—file will only be accessed sequentially,
  - MPI\_MODE\_APPEND—set initial position of all file pointers to end of file.

Advice to users. C users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (End of advice to users.)

Advice to implementors. The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End of advice to implementors.*)

The modes MPI\_MODE\_RDONLY, MPI\_MODE\_RDWR, MPI\_MODE\_WRONLY, MPI\_MODE\_CREATE, and MPI\_MODE\_EXCL have identical semantics to their POSIX counterparts [44]. Exactly one of MPI\_MODE\_RDONLY, MPI\_MODE\_RDWR, or MPI\_MODE\_WRONLY, must be specified. It is erroneous to specify MPI\_MODE\_CREATE or MPI\_MODE\_EXCL in conjunction with MPI\_MODE\_RDONLY; it is erroneous to specify MPI\_MODE\_SEQUENTIAL together with MPI\_MODE\_RDWR.

The MPI\_MODE\_DELETE\_ON\_CLOSE mode causes the file to be deleted (equivalent to performing an MPI\_FILE\_DELETE) when the file is closed.

The MPI\_MODE\_UNIQUE\_OPEN mode allows an implementation to optimize access by eliminating the overhead of file locking. It is erroneous to open a file in this mode unless the file will not be concurrently opened elsewhere.

Advice to users. For MPI\_MODE\_UNIQUE\_OPEN, not opened elsewhere includes both inside and outside the MPI environment. In particular, one needs to be aware of potential external events which may open files (e.g., automated backup facilities). When MPI\_MODE\_UNIQUE\_OPEN is specified, the user is responsible for ensuring that no such external events take place. (End of advice to users.)

The MPI\_MODE\_SEQUENTIAL mode allows an implementation to optimize access to some sequential devices (tapes and network streams). It is erroneous to attempt nonsequential access to a file that has been opened in this mode.

Specifying MPI\_MODE\_APPEND only guarantees that all shared and individual file pointers are positioned at the initial end of file when MPI\_FILE\_OPEN returns. Subsequent positioning of file pointers is application dependent. In particular, the implementation does not ensure that all writes are appended.

Errors related to the access mode are raised in the class MPI\_ERR\_AMODE.

The info argument is used to provide information regarding file access patterns and file system specifics (see Section 14.2.8). The constant MPI\_INFO\_NULL can be used when no info needs to be specified.

Advice to users. Some file attributes are inherently implementation dependent (e.g., file permissions). These attributes must be set using either the info argument or facilities outside the scope of MPI. (End of advice to users.)

Files are opened by default using nonatomic mode file consistency semantics (see Section 14.6.1). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI\_FILE\_SET\_ATOMICITY.

 $\overline{7}$ 

 $^{24}$ 

```
1
     14.2.2 Closing a File
\mathbf{2}
3
4
     MPI_FILE_CLOSE(fh)
5
       INOUT
                 fh
                                              file handle (handle)
6
7
     C binding
8
     int MPI_File_close(MPI_File *fh)
9
10
     Fortran 2008 binding
11
     MPI_File_close(fh, ierror)
12
          TYPE(MPI_File), INTENT(INOUT) :: fh
13
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
     Fortran binding
15
     MPI_FILE_CLOSE(FH, IERROR)
16
17
          INTEGER FH, IERROR
18
          MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
19
     MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
20
     opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
21
     MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.
22
23
           Advice to users. If the file is deleted on close, and there are other processes currently
^{24}
           accessing the file, the status of the file and the behavior of future accesses by these
25
           processes are implementation dependent. (End of advice to users.)
26
27
          The user is responsible for ensuring that all outstanding nonblocking requests and
28
     split collective operations associated with fh made by a process have completed before that
29
     process calls MPI_FILE_CLOSE.
30
          The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
^{31}
     MPI_FILE_NULL.
32
33
     14.2.3 Deleting a File
34
35
36
     MPI_FILE_DELETE(filename, info)
37
       IN
                 filename
                                              name of file to delete (string)
38
39
       IN
                 info
                                              info object (handle)
40
41
     C binding
42
     int MPI_File_delete(const char *filename, MPI_Info info)
43
     Fortran 2008 binding
44
     MPI_File_delete(filename, info, ierror)
45
46
          CHARACTER(LEN=*), INTENT(IN) :: filename
47
          TYPE(MPI_Info), INTENT(IN) :: info
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

#### Fortran binding

MPI_FILE_DELETE(F	ILENAME, INFO,	IERROR)
CHARACTER*(*)	FILENAME	
INTEGER INFO,	IERROR	

MPI\_FILE\_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI\_FILE\_DELETE raises an error in the class MPI\_ERR\_NO\_SUCH\_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 14.2.8). The constant MPI\_INFO\_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI\_ERR\_FILE\_IN\_USE or MPI\_ERR\_ACCESS will be raised. Errors are raised using the default file error handler (see Section 14.7).

## 14.2.4 Resizing a File

 MPI\_FILE\_SET\_SIZE(fh, size)

 INOUT
 fh

 IN
 size

 size
 size to truncate or expand file (integer)

#### C binding

int MPI\_File\_set\_size(MPI\_File fh, MPI\_Offset size)

## Fortran 2008 binding

MPI\_File\_set\_size(fh, size, ierror)
 TYPE(MPI\_File), INTENT(IN) :: fh
 INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: size
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

### Fortran binding

```
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

MPI\_FILE\_SET\_SIZE resizes the file associated with the file handle fh. size is measured in bytes from the beginning of the file. MPI\_FILE\_SET\_SIZE is collective; all processes in the group must pass identical values for size.

If size is smaller than the current file size, the file is truncated at the position defined by size. The implementation is free to deallocate file blocks located beyond this position.

If size is larger than the current file size, the file size becomes size. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and size) are undefined. It is implementation dependent whether the MPI\_FILE\_SET\_SIZE routine allocates file space—use MPI\_FILE\_PREALLOCATE to force file space to be reserved.

MPI\_FILE\_SET\_SIZE does not affect the individual file pointers or the shared file

1 pointer. If MPI\_MODE\_SEQUENTIAL mode was specified when the file was opened, it is  $\mathbf{2}$ erroneous to call this routine. 3 4 Advice to users. It is possible for the file pointers to point beyond the end of file after a MPI\_FILE\_SET\_SIZE operation truncates a file. This is valid, and equivalent 5to seeking beyond the current end of file. (End of advice to users.) 6 7 All nonblocking requests and split collective operations on fh must be completed before 8 calling MPI\_FILE\_SET\_SIZE. Otherwise, calling MPI\_FILE\_SET\_SIZE is erroneous. As far 9 as consistency semantics are concerned, MPI\_FILE\_SET\_SIZE is a write operation that 10 conflicts with operations that access bytes at displacements between the old and new file 11 sizes (see Section 14.6.1). 1213 14.2.5 Preallocating Space for a File 14151617MPI\_FILE\_PREALLOCATE(fh, size) 18 INOUT fh file handle (handle) 19IN size to preallocate file (integer) size 202122C binding 23int MPI\_File\_preallocate(MPI\_File fh, MPI\_Offset size)  $^{24}$ Fortran 2008 binding 25MPI\_File\_preallocate(fh, size, ierror) 26TYPE(MPI\_File), INTENT(IN) :: fh 27INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(IN) :: size 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2930 Fortran binding  $^{31}$ MPI\_FILE\_PREALLOCATE(FH, SIZE, IERROR) 32 INTEGER FH, IERROR 33 INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE 34MPI\_FILE\_PREALLOCATE ensures that storage space is allocated for the first size bytes 35 of the file associated with fh. MPI\_FILE\_PREALLOCATE is collective; all processes in the 36 group must pass identical values for size. Regions of the file that have previously been 37 written are unaffected. For newly allocated regions of the file, MPI\_FILE\_PREALLOCATE 38 has the same effect as writing undefined data. If size is larger than the current file size, the 39 file size increases to size. If size is less than or equal to the current file size, the file size is 40 unchanged. 41 The treatment of file pointers, pending nonblocking accesses, and file consistency is the 42same as with MPI\_FILE\_SET\_SIZE. If MPI\_MODE\_SEQUENTIAL mode was specified when 43 the file was opened, it is erroneous to call this routine. 4445Advice to users. In some implementations, file preallocation may be time-consuming. 46 (End of advice to users.) 47

14.2.6 Querying the Size of a File 1  $\mathbf{2}$ 3 4 MPI\_FILE\_GET\_SIZE(fh, size) 5 IN fh file handle (handle) 6 OUT size of the file in bytes (integer) size 7 8 9 C binding 10 int MPI\_File\_get\_size(MPI\_File fh, MPI\_Offset \*size) 11 Fortran 2008 binding 12MPI\_File\_get\_size(fh, size, ierror) 13 TYPE(MPI\_File), INTENT(IN) :: fh 14INTEGER(KIND=MPI\_OFFSET\_KIND), INTENT(OUT) :: size 15INTEGER, OPTIONAL, INTENT(OUT) :: ierror 1617 Fortran binding 18 MPI\_FILE\_GET\_SIZE(FH, SIZE, IERROR) 19 INTEGER FH, IERROR 20INTEGER(KIND=MPI\_OFFSET\_KIND) SIZE 21MPI\_FILE\_GET\_SIZE returns, in size, the current size in bytes of the file associated with 22 the file handle fh. As far as consistency semantics are concerned, MPI\_FILE\_GET\_SIZE is a 23data access operation (see Section 14.6.1). 242514.2.7 Querying File Parameters 262728 MPI\_FILE\_GET\_GROUP(fh, group) 29 30 IN fh file handle (handle) 31OUT group which opened the file (handle) group 32 33 C binding 34 int MPI\_File\_get\_group(MPI\_File fh, MPI\_Group \*group) 35 36 Fortran 2008 binding 37 MPI\_File\_get\_group(fh, group, ierror) 38 TYPE(MPI\_File), INTENT(IN) :: fh 39 TYPE(MPI\_Group), INTENT(OUT) :: group 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 42Fortran binding MPI\_FILE\_GET\_GROUP(FH, GROUP, IERROR) 43 44INTEGER FH, GROUP, IERROR 45MPI\_FILE\_GET\_GROUP returns a duplicate of the group of the communicator used to 46

MPI\_FILE\_GET\_GROUP returns a duplicate of the group of the communicator used to open the file associated with fh. The group is returned in group. The user is responsible for freeing group.

47

```
1
     MPI_FILE_GET_AMODE(fh, amode)
\mathbf{2}
       IN
                 fh
                                             file handle (handle)
3
       OUT
                 amode
                                            file access mode used to open the file (integer)
4
5
     C binding
6
\overline{7}
     int MPI_File_get_amode(MPI_File fh, int *amode)
8
     Fortran 2008 binding
9
     MPI_File_get_amode(fh, amode, ierror)
10
          TYPE(MPI_File), INTENT(IN) :: fh
11
          INTEGER, INTENT(OUT) :: amode
12
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     Fortran binding
     MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
15
16
          INTEGER FH, AMODE, IERROR
17
         MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with
18
     fh.
19
     Example 14.1 In Fortran 77, decoding an amode bit vector will require a routine such as
20
21
     the following:
22
     SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
23
      !
24
      !
          TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
25
          IF SET, RETURN 1 IN BIT_FOUND, O OTHERWISE
      !
26
      !
27
          INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
28
          BIT_FOUND = 0
29
          CP\_AMODE = AMODE
30
     100 CONTINUE
^{31}
          LBIT = 0
32
          HIFOUND = 0
33
          DO L = MAX_BIT, O, -1
34
             MATCHER = 2**L
35
             IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
36
                  HIFOUND = 1
37
                 LBIT = MATCHER
38
                 CP_AMODE = CP_AMODE - MATCHER
39
             END IF
40
          END DO
41
          IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
42
          IF (BIT_FOUND .EQ. O .AND. HIFOUND .EQ. 1 .AND. &
43
              CP_AMODE .GT. 0) GO TO 100
44
     END
45
46
     This routine could be called successively to decode amode, one bit at a time. For example,
47
      the following code fragment would check for MPI_MODE_RDONLY.
48
```

```
CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF
```

# 14.2.8 File Info

Hints specified via info (see Chapter 10) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI\_FILE\_GET\_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per file basis, in MPI\_FILE\_OPEN, MPI\_FILE\_DELETE, MPI\_FILE\_SET\_VIEW, and MPI\_FILE\_SET\_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI\_FILE\_SET\_VIEW or MPI\_FILE\_SET\_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

Advice to implementors. It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored.

However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

MPI_FILE	_SET_INFO(fh, info)	
INOUT	fh	file handle (handle)
IN	info	info object (handle)

# C binding

int MPI\_File\_set\_info(MPI\_File fh, MPI\_Info info)

```
Fortran 2008 binding
```

MPI\_File\_set\_info(fh, info, ierror)
 TYPE(MPI\_File), INTENT(IN) :: fh
 TYPE(MPI\_Info), INTENT(IN) :: info
 INTEGER, OPTIONAL, INTENT(OUT) :: ierror

## Fortran binding

MPI\_FILE\_SET\_INFO(FH, INFO, IERROR) INTEGER FH, INFO, IERROR

MPI\_FILE\_SET\_INFO updates the hints of the file associated with fh using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not

1 specified by info. It also has no effect on previously set or defaulted hints that are specified  $\mathbf{2}$ by info, but are ignored by the MPI implementation in this call to MPI\_FILE\_SET\_INFO. 3 MPI\_FILE\_SET\_INFO is a collective routine. The info object may be different on each 4 process, but any info entries that an implementation requires to be the same on all processes  $\mathbf{5}$ must appear with the same value in each process's info object. 6  $\overline{7}$ Advice to users. Many info items that an implementation can use when it creates or opens a file cannot easily be changed once the file has been created or opened. Thus, 8 an implementation may ignore hints issued in this call that it would have accepted in 9 an open call. An implementation may also be unable to update certain info hints in a 10 call to MPI\_FILE\_SET\_VIEW or MPI\_FILE\_SET\_INFO. MPI\_FILE\_GET\_INFO can be 11 used to determine whether info changes were ignored by the implementation. (End of 12advice to users.) 13 141516MPI\_FILE\_GET\_INFO(fh, info\_used) 17IN fh file handle (handle) 18 19info\_used OUT new info object (handle) 2021C binding 22int MPI\_File\_get\_info(MPI\_File fh, MPI\_Info \*info\_used) 23Fortran 2008 binding  $^{24}$ 25MPI\_File\_get\_info(fh, info\_used, ierror) 26TYPE(MPI\_File), INTENT(IN) :: fh TYPE(MPI\_Info), INTENT(OUT) :: info\_used 27INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2829Fortran binding 30 MPI\_FILE\_GET\_INFO(FH, INFO\_USED, IERROR)  $^{31}$ INTEGER FH, INFO\_USED, IERROR 32 33 MPI\_FILE\_GET\_INFO returns a new info object containing the hints of the file associ-34ated with fh. The current setting of all hints related to this file is returned in info\_used. An 35 MPI implementation is required to return all hints that are supported by the implementa-36 tion and have default values specified; any user-supplied hints that were not ignored by the 37 implementation; and any additional hints that were set by the implementation. If no such 38hints exist, a handle to a newly created info object is returned that contains no (key, value) 39 pairs. The user is responsible for freeing info\_used via MPI\_INFO\_FREE. 40 $^{41}$ Reserved File Hints 42Some potentially useful hints (info key values) are outlined below. The following key values 43 are reserved. An implementation is not required to interpret these key values, but if it does 44interpret the key value, it must provide the functionality described. (For more details on 45"info," see Chapter 10.) 46 These hints mainly affect access patterns and the layout of data on parallel I/O devices. 47

 $_{48}$  For each hint name introduced, we describe the purpose of the hint, and the type of the hint

value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., "file\_perm" is only useful during file creation).

- "access\_style" (comma separated list of strings): This hint specifies the manner in which the file will be accessed until the file is closed or until the "access\_style" key value is altered. The hint value is a comma separated list of the following: "read\_once", "write\_once", "read\_mostly", "write\_mostly", "sequential", "reverse\_sequential", and "random".
- "collective\_buffering" (boolean) [SAME]: This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are "true" and "false". Collective buffering parameters are further directed via additional hints: "cb\_block\_size", "cb\_buffer\_size", and "cb\_nodes".
- "cb\_block\_size" (integer) [SAME]: This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.
- "cb\_buffer\_size" (integer) [SAME]: This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of "cb\_block\_size".
- "cb\_nodes" (integer) [SAME]: This hint specifies the number of target nodes to be used for collective buffering.
- "chunked" (comma separated list of integers) [SAME]: This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).
- "chunked\_item" (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.
- "chunked\_size" (comma separated list of integers) [SAME]: This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.
- "filename" (string): This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI\_FILE\_GET\_INFO. This key is ignored when passed to MPI\_FILE\_OPEN, MPI\_FILE\_SET\_VIEW, MPI\_FILE\_SET\_INFO, and MPI\_FILE\_DELETE.
- "file\_perm" (string) [SAME]: This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI\_FILE\_OPEN with an amode

 $\overline{7}$ 

 $^{31}$ 

1 2		hat includes MPI_M ion dependent.	ODE_CREATE. The set of valid values for this key is implementa-
3 4 5 6	0		eparated list of strings) [SAME]: This hint specifies the list should be used to store the file. This hint is most relevant when
7 8 9 10	W		<b>ME</b> ]: This hint specifies the number of parallel processes that igned to run programs that access this file. This hint is most e is created.
11 12			<b>[SAME]:</b> This hint specifies the number of I/O devices in the most relevant when the file is created.
13 14 15			) [SAME]: This hint specifies the number of I/O devices that riped across, and is relevant only when the file is created.
16 17 18 19 20	u I,	sed for this file. Th /O device before p	<b>[SAME]:</b> This hint specifies the suggested striping unit to be the striping unit is the amount of consecutive data assigned to one rogressing to the next device, when striping across a number of sed in bytes. This hint is relevant only when the file is created.
21 22 23 24	14.3	File Views	
25	MPI_F	ILE_SET_VIEW(fh,	disp, etype, filetype, datarep, info)
26	INOU	T fh	file handle (handle)
27 28	IN	disp	displacement (integer)
29	IN	etype	elementary datatype (handle)
30	IN	filetype	filetype (handle)
31 32	IN	datarep	data representation (string)
33 34	IN	info	info object (handle)
35 36 37	C bind int MP	PI_File_set_view(	(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, type filetype, const char *datarep, MPI_Info info)
<ol> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> <li>46</li> </ol>	MPI_Fi TY IN TY CH TY IN	PE(MPI_File), IN TEGER(KIND=MPI_C PE(MPI_Datatype) MARACTER(LEN=*), PE(MPI_Info), IN TEGER, OPTIONAL,	<pre>disp, etype, filetype, datarep, info, ierror) ITENT(IN) :: fh DFFSET_KIND), INTENT(IN) :: disp , INTENT(IN) :: etype, filetype INTENT(IN) :: datarep ITENT(IN) :: info INTENT(OUT) :: ierror</pre>
47 48		n binding LE_SET_VIEW(FH,	DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)

# INTEGER FH, ETYPE, FILETYPE, INFO, IERROR INTEGER(KIND=MPI\_OFFSET\_KIND) DISP CHARACTER\*(\*) DATAREP

The MPI\_FILE\_SET\_VIEW routine changes the process's view of the data in the file. The start of the view is set to disp; the type of data is set to etype; the distribution of data to processes is set to filetype; and the representation of data in the file is set to datarep. In addition, MPI\_FILE\_SET\_VIEW resets the individual file pointers and the shared file pointer to zero. MPI\_FILE\_SET\_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 14.5.1 for further details.

If MPI\_MODE\_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI\_DISPLACEMENT\_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI\_DISPLACEMENT\_CURRENT is invalid unless the amode for the file has MPI\_MODE\_SEQUENTIAL set.

Rationale. For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI\_DISPLACEMENT\_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

Advice to implementors. It is expected that a call to MPI\_FILE\_SET\_VIEW will immediately follow MPI\_FILE\_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 14.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

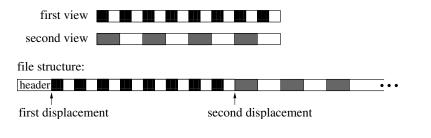


Figure 14.3: Displacements

 $\mathbf{2}$ 

 $^{24}$ 

<sup>(</sup>End of advice to users.)

An *etype* (*elementary* datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

> Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the etype (see Section 14.5). (End of advice to users.)

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different processes may still overlap each other.

If a filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the disp, etype, and filetype arguments can be changed via future calls to MPI\_FILE\_SET\_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype. The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 14.2.8). The constant MPI\_INFO\_NULL refers to the null info and can be used when no info needs to be specified.

The datarep argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 14.5) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI\_FILE\_SET\_VIEW—otherwise, the call to MPI\_FILE\_SET\_VIEW is erroneous.

31 32 33

34

42

29

30

MPI\_FILE\_GET\_VIEW(fh, disp, etype, filetype, datarep)

35	IN	fh	file handle (handle)
36	OUT	disp	displacement (integer)
37 38	OUT	etype	elementary datatype (handle)
39	OUT	filetype	filetype (handle)
40	OUT	datarep	data representation (string)
41			

```
C binding
```

```
43 int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
44 MPI_Datatype *filetype, char *datarep)
45
```

```
46 Fortran 2008 binding
```

```
    MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
```

7

8

9

10 11

12

13

14

```
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
CHARACTER(LEN=*), INTENT(OUT) :: datarep
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
Fortran binding
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
```

```
INTEGER FH, ETYPE, FILETYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) DISP
CHARACTER*(*) DATAREP
```

MPI\_FILE\_GET\_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI\_MAX\_DATAREP\_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI\_FILE\_GET\_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

# 14.4 Data Access

# 14.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 14.1.

positioning	synchronism	cod	ordination
		noncollective	collective
explicit	blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
offsets		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
	nonblocking	MPI_FILE_IREAD_AT	MPI_FILE_IREAD_AT_ALL
		MPI_FILE_IWRITE_AT	MPI_FILE_IWRITE_AT_ALL
	split collective	N/A	MPI_FILE_READ_AT_ALL_BEGIN
			MPI_FILE_READ_AT_ALL_END
			MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END
individual	blocking	MPI_FILE_READ	MPI_FILE_READ_ALL
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL
	nonblocking	MPI_FILE_IREAD	MPI_FILE_IREAD_ALL
		MPI_FILE_IWRITE	MPI_FILE_IWRITE_ALL
	split collective	N/A	MPI_FILE_READ_ALL_BEGIN
			MPI_FILE_READ_ALL_END
			MPI_FILE_WRITE_ALL_BEGIN
			MPI_FILE_WRITE_ALL_END
shared	blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED
	nonblocking	MPI_FILE_IREAD_SHARED	N/A
		MPI_FILE_IWRITE_SHARED	
	split collective	N/A	MPI_FILE_READ_ORDERED_BEGIN
			MPI_FILE_READ_ORDERED_END
			MPI_FILE_WRITE_ORDERED_BEGIN
			MPI_FILE_WRITE_ORDERED_END

Table 14.1: Data access routines

 $\mathbf{2}$ 

 $\frac{4}{5}$ 

POSIX read()/fread() and write()/fwrite() are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI\_FILE\_READ and MPI\_FILE\_WRITE. Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user's buffer after a read operation completes. For writes, however, the MPI\_FILE\_SYNC routine provides the only guarantee that data has been transferred to the storage device.

<sup>9</sup> Positioning 10

MPI provides three types of positioning for data access routines: explicit offsets, individual file pointers, and shared file pointers. The different positioning methods may be mixed within the same program and do not affect each other.

The data access routines that accept explicit offsets contain \_AT in their name (e.g., MPI\_FILE\_WRITE\_AT). Explicit offset operations perform data access at the file position given directly as an argument—no file pointer is used nor updated. Note that this is not equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. Operations with explicit offsets are described in Section 14.4.2.

The names of the individual file pointer routines contain no positional qualifier (e.g., MPI\_FILE\_WRITE). Operations with individual file pointers are described in Section 14.4.3. The data access routines that use shared file pointers contain \_SHARED or \_ORDERED in their name (e.g., MPI\_FILE\_WRITE\_SHARED). Operations with shared file pointers are described in Section 14.4.4.

The main semantic issues with MPI-maintained file pointers are how and when they are updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In a nonblocking or split collective operation, the pointer is updated by the call that initiates the I/O, possibly before the access completes.

More formally,

32

37

$$new\_file\_offset = old\_file\_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

where *count* is the number of *datatype* items to be accessed, elements(X) is the number of predefined datatypes in the typemap of X, and *old\_file\_offset* is the value of the implicit offset before the call. The file position, *new\_file\_offset*, is in terms of a count of etypes relative to the current view.

<sup>38</sup> Synchronism

 $_{40}^{39}$  MPI supports blocking and nonblocking I/O routines.

A blocking I/O call will not return until the I/O request is completed.

A nonblocking I/O call with not return tilter the I/O request is completed. A nonblocking I/O call initiates an I/O operation, but does not wait for it to complete. Given suitable hardware, this allows the transfer of data out of and into the user's buffer to proceed concurrently with computation. A separate request complete call (MPI\_WAIT, MPI\_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm that the data has been read or written and that it is safe for the user to reuse the buffer. The nonblocking versions of the routines are named MPI\_FILE\_IXXX, where the I stands for immediate.

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

7

 $\mathbf{2}$ 

 $^{24}$ 

 $^{31}$ 

 $\frac{43}{44}$ 

It is erroneous to access the local buffer of a nonblocking data access operation, or to use that buffer as the source or target of other communications, between the initiation and completion of the operation.

The split collective routines support a restricted form of "nonblocking" operations for collective data access (see Section 14.4.5).

#### Coordination

Every noncollective data access routine MPI\_FILE\_XXX has a collective counterpart. For most routines, this counterpart is MPI\_FILE\_XXX\_ALL or a pair of MPI\_FILE\_XXX\_BEGIN and MPI\_FILE\_XXX\_END. The counterparts to the MPI\_FILE\_XXX\_SHARED routines are MPI\_FILE\_XXX\_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 14.6.4 for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

#### Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 5.1.11. The data is accessed from those parts of the file specified by the current view (Section 14.3). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI\_TEST, MPI\_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 19.1.10–19.1.20. (End of advice to users.)

1 For blocking routines, status is returned directly. For nonblocking routines and split  $\mathbf{2}$ collective routines, status is returned when the operation is completed. The number of 3 datatype entries and predefined elements accessed by the calling process can be extracted 4 from status by using MPI\_GET\_COUNT and MPI\_GET\_ELEMENTS (or  $\mathbf{5}$ MPI\_GET\_ELEMENTS\_X), respectively. The interpretation of the MPI\_ERROR field is the

6 same as for other operations—normally undefined, but meaningful if an MPI routine returns  $\overline{7}$ MPI\_ERR\_IN\_STATUS. The user can pass (in C and Fortran) MPI\_STATUS\_IGNORE in the 8 status argument if the return value of this argument is not needed. The status can be 9 passed to MPI\_TEST\_CANCELLED to determine if the operation was cancelled. All other 10 fields of status are undefined.

11When reading, a program can detect the end of file by noting that the amount of data 12read is less than the amount requested. Writing past the end of file increases the file size. 13The amount of data accessed will be the amount requested, unless an error is raised (or a 14read reaches the end of file).

1516

17

14.4.2 Data Access with Explicit Offsets

If MPI\_MODE\_SEQUENTIAL mode was specified when the file was opened, it is erroneous to 18 call the routines in this section. 19

2021

32

33

3435

36

37

38

39

40

41

42

43

MPI\_FILE\_READ\_AT(fh, offset, buf, count, datatype, status)

22			it, additype, status)
23	IN	fh	file handle (handle)
24	IN	offset	file offset (integer)
25 26	OUT	buf	initial address of buffer (choice)
20 27	IN	count	number of elements in buffer (integer)
28	IN	datatype	datatype of each buffer element (handle)
29	OUT	status	status object (status)
30			
31	C binding	7	

```
C binding
```

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

```
int MPI_File_read_at_c(MPI_File fh, MPI_Offset offset, void *buf,
             MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
```

```
Fortran 2008 binding
```

```
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
   TYPE(MPI_File), INTENT(IN) :: fh
   INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
```

```
TYPE(*), DIMENSION(..) :: buf
INTEGER, INTENT(IN) :: count
```

```
TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
44
         TYPE(MPI_Status) :: status
```

```
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
46
```

```
47
     MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) !(_c)
48
         TYPE(MPI_File), INTENT(IN) :: fh
```

INTE	GER(KIND=MPI_OFFSE	T_KIND), INTENT(IN) :: offset	1
	E(*), DIMENSION()		2
		_KIND), INTENT(IN) :: count	3
		TENT(IN) :: datatype	4 5
	E(MPI_Status) :: st EGER, OPTIONAL, INT		6
			7
Fortran	•		8
		T, BUF, COUNT, DATATYPE, STATUS, IERROR) ATYPE, STATUS(MPI_STATUS_SIZE), IERROR	9
	GER (KIND=MPI_OFFSE		10
	be> BUF(*)		11
			12 13
MPL	_FILE_READ_AI reads	s a file beginning at the position specified by offset.	13
			15
MPI_FILE	E_READ_AT_ALL(fh, c	ffset, buf, count, datatype, status)	16
IN	fh	file handle (handle)	17
IN	offset	file offset (integer)	18
OUT	buf	initial address of buffer (choice)	19
			20 21
IN	count	number of elements in buffer (integer)	21
IN	datatype	datatype of each buffer element (handle)	23
OUT	status	status object (status)	24
			25
C bindi	0		26
int MPI_		PI_File fh, MPI_Offset offset, void *buf,	27
	int count, MP	I_Datatype datatype, MPI_Status *status)	28 29
int MPI_	File_read_at_all_c	(MPI_File fh, MPI_Offset offset, void *buf,	30
	MPI_Count cou	nt, MPI_Datatype datatype, MPI_Status *status)	31
Fortran	2008 binding		32
		ffset, buf, count, datatype, status, ierror)	33
	E(MPI_File), INTENT		34
		T_KIND), INTENT(IN) :: offset	35
	E(*), DIMENSION()		36 37
	EGER, INTENT(IN) ::		38
	E(MPI_Datatype), IN E(MPI_Status) :: st	TENT(IN) :: datatype	39
	EGER, OPTIONAL, INT		40
			41
MP1_File	e_read_at_all(in, or !(_c)	ffset, buf, count, datatype, status, ierror)	42
турі	:(_C) E(MPI_File), INTENT	(IN) ·· fh	43
	-	T_KIND), INTENT(IN) :: offset	44
	E(*), DIMENSION()		45 46
INTE	GER(KIND=MPI_COUNT	_KIND), INTENT(IN) :: count	40
TYPE	(MPI_Datatype), IN	TENT(IN) :: datatype	48

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     Fortran binding
4
     MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
5
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
6
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
7
         <type> BUF(*)
8
9
         MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT
10
     interface.
11
12
     MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
13
14
       INOUT
                fh
                                           file handle (handle)
15
       IN
                offset
                                           file offset (integer)
16
       IN
                buf
                                           initial address of buffer (choice)
17
18
       IN
                count
                                           number of elements in buffer (integer)
19
       IN
                                           datatype of each buffer element (handle)
                datatype
20
       OUT
                                           status object (status)
                status
21
22
     C binding
23
     int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
^{24}
                    int count, MPI_Datatype datatype, MPI_Status *status)
25
26
     int MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
27
                    MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
28
29
     Fortran 2008 binding
30
     MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
         TYPE(MPI_File), INTENT(IN) :: fh
31
32
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
33
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
34
         INTEGER, INTENT(IN) :: count
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Status) :: status
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) !(_c)
39
         TYPE(MPI_File), INTENT(IN) :: fh
40
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
41
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         TYPE(MPI_Status) :: status
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     Fortran binding
48
     MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
```

INTE	GER FH, COUNT, DAT GER(KIND=MPI_OFFSE be> BUF(*)	TATYPE, STATUS(MPI_STATUS_SIZE), IERROR ET_KIND) OFFSET	1 2 3
• -		the effect of the maritime energies a loss offert	4
IMPI_	FILE_WRITE_AT wri	tes a file beginning at the position specified by offset.	5
MPI FILF	WRITE AT ALL(fh	offset, buf, count, datatype, status)	6 7
INOUT	fh	file handle (handle)	8
			9 10
IN	offset	file offset (integer)	10
IN	buf	initial address of buffer (choice)	12
IN	count	number of elements in buffer (integer)	13
IN	datatype	datatype of each buffer element (handle)	14
OUT	status	status object (status)	15 16
C bindin int MPI_	File_write_at_all(	(MPI_File fh, MPI_Offset offset, const void *buf, PI_Datatype datatype, MPI_Status *status)	17 18 19 20
int MPI_		_c(MPI_File fh, MPI_Offset offset, buf, MPI_Count count, MPI_Datatype datatype, status)	21 22 23
MPI_File TYPE INTE TYPE INTE TYPE TYPE	C(MPI_File), INTENT GER(KIND=MPI_OFFSE C(*), DIMENSION() GER, INTENT(IN) :: C(MPI_Datatype), IN C(MPI_Status) :: st	ET_KIND), INTENT(IN) :: offset ), INTENT(IN) :: buf : count NTENT(IN) :: datatype	24 25 26 27 28 29 30 31 32 33
TYPE INTE TYPE INTE TYPE TYPE	!(_c) :(MPI_File), INTENT :GER(KIND=MPI_OFFSE :(*), DIMENSION() :GER(KIND=MPI_COUNT :(MPI_Datatype), IN :(MPI_Status) :: st	ET_KIND), INTENT(IN) :: offset ), INTENT(IN) :: buf [_KIND), INTENT(IN) :: count NTENT(IN) :: datatype	34 35 36 37 38 39 40 41 42 43
Fortran	binding		43 44
	-	OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)	45
		TATYPE, STATUS(MPI_STATUS_SIZE), IERROR	46
	CGER(KIND=MPI_OFFSE >e> BUF(*)	"I_VIND) OLEDEI	47
. Cyb			48

```
1
         MPI_FILE_WRITE_AT_ALL is a collective version of the blocking
\mathbf{2}
     MPI_FILE_WRITE_AT interface.
3
4
     MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
5
6
       IN
                fh
                                            file handle (handle)
7
       IN
                offset
                                            file offset (integer)
8
       OUT
                buf
                                           initial address of buffer (choice)
9
10
       IN
                count
                                           number of elements in buffer (integer)
11
                                           datatype of each buffer element (handle)
       IN
                datatype
12
       OUT
                request
                                           request object (handle)
13
14
     C binding
15
16
     int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
17
                    MPI_Datatype datatype, MPI_Request *request)
18
     int MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf,
19
                    MPI_Count count, MPI_Datatype datatype, MPI_Request *request)
20
     Fortran 2008 binding
21
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
22
23
         TYPE(MPI_File), INTENT(IN) :: fh
^{24}
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
25
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
26
         INTEGER, INTENT(IN) :: count
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
^{31}
         TYPE(MPI_File), INTENT(IN) :: fh
32
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
33
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
34
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     Fortran binding
40
     MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
41
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
42
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
43
         <type> BUF(*)
44
         MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.
45
46
47
48
```

MPI_FILE	_IREAD_AT_ALL(fh, offset, b	uf, count, datatype, request)	1
IN	fh	file handle (handle)	2 3
IN	offset	file offset (integer)	4
OUT	buf	initial address of buffer (choice)	5
IN	count	number of elements in buffer (integer)	6
IN	datatype	datatype of each buffer element (handle)	7 8
OUT	request	request object (handle)	9
			10
C binding int MPI_F	File_iread_at_all(MPI_Fil	e fh, MPI_Offset offset, void *buf, type datatype, MPI_Request *request)	11 12 13 14
int MPI_F		Tile fh, MPI_Offset offset, void *buf, I_Datatype datatype, MPI_Request *request)	15 16
MPI_File_ TYPE( INTEC TYPE( INTEC TYPE( TYPE(	2008 binding _iread_at_all(fh, offset, (MPI_File), INTENT(IN) :: SER(KIND=MPI_OFFSET_KIND) (*), DIMENSION(), ASYNC SER, INTENT(IN) :: count (MPI_Datatype), INTENT(IN (MPI_Request), INTENT(OUT SER, OPTIONAL, INTENT(OUT	), INTENT(IN) :: offset CHRONOUS :: buf I) :: datatype C) :: request	17 18 19 20 21 22 23 24 25 26
TYPE( INTEC TYPE( INTEC TYPE( TYPE)	_iread_at_all(fh, offset,	), INTENT(IN) :: offset CHRONOUS :: buf INTENT(IN) :: count I) :: datatype C) :: request	20 27 28 29 30 31 32 33 34 35 36
INTEC INTEC	0		30 37 38 39 40 41
		nblocking version of MPI_FILE_READ_AT_ALL. See king collective file operations.	41 42 43 44 45 46 47

```
1
     MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
2
       INOUT
                fh
                                           file handle (handle)
3
                offset
       IN
                                           file offset (integer)
4
5
                buf
       IN
                                           initial address of buffer (choice)
6
       IN
                count
                                           number of elements in buffer (integer)
7
       IN
                datatype
                                           datatype of each buffer element (handle)
8
9
       OUT
                request
                                           request object (handle)
10
11
     C binding
12
     int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
13
                    int count, MPI_Datatype datatype, MPI_Request *request)
14
     int MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
15
                   MPI_Count count, MPI_Datatype datatype, MPI_Request *request)
16
17
     Fortran 2008 binding
18
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
19
         TYPE(MPI_File), INTENT(IN) :: fh
20
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
21
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
22
         INTEGER, INTENT(IN) :: count
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
27
         TYPE(MPI_File), INTENT(IN) :: fh
28
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
29
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     Fortran binding
36
     MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
37
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
38
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
39
         <type> BUF(*)
40
         MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.
41
42
43
44
45
46
47
48
```

MPI_FILE	_IWRITE_AT_ALL(fh, offset, b	uf, count, datatype, request)	1
INOUT	fh	file handle (handle)	$\frac{2}{3}$
IN	offset	file offset (integer)	4
IN	buf	initial address of buffer (choice)	5
IN	count	number of elements in buffer (integer)	6
IN	datatype	datatype of each buffer element (handle)	7 8
OUT	request	request object (handle)	9
			10
C bindin	•		11 12
int MPI_F		<pre>le fh, MPI_Offset offset, const void *buf, ype datatype, MPI_Request *request)</pre>	13
			14
int MPI_F		File fh, MPI_Offset offset,	15
	MPI_Request *request	_Count count, MPI_Datatype datatype,	16 17
Distance (		, ,	18
	2008 binding	, buf, count, datatype, request, ierror)	19
	(MPI_File), INTENT(IN) ::	vi i	20
INTEC	SER(KIND=MPI_OFFSET_KIND)	, INTENT(IN) :: offset	21
		Γ(IN), ASYNCHRONOUS :: buf	22 23
	GER, INTENT(IN) :: count		24
	(MPI_Datatype), INTENT(IN)		25
	(MPI_Request), INTENT(OUT) GER, OPTIONAL, INTENT(OUT)	-	26
			27
MP1_File_	_iwrite_at_all(ih, offset) !(_c)	, buf, count, datatype, request, ierror)	28 29
TYPE	(MPI_File), INTENT(IN) ::	fh	30
	GER(KIND=MPI_OFFSET_KIND)		31
TYPE	(*), DIMENSION(), INTEN	Γ(IN), ASYNCHRONOUS :: buf	32
	<pre>SER(KIND=MPI_COUNT_KIND),</pre>		33
	(MPI_Datatype), INTENT(IN)		34
	(MPI_Request), INTENT(OUT) GER, OPTIONAL, INTENT(OUT)	-	35 36
			30
Fortran k	0		38
	_IWRIIE_AI_ALL(FH, UFFSEI) GER FH, COUNT, DATATYPE, H	, BUF, COUNT, DATATYPE, REQUEST, IERROR)	39
	GER(KIND=MPI_OFFSET_KIND)		40
	<pre>e&gt; BUF(*)</pre>		41
• -		nblocking version of MPI_FILE_WRITE_AT_ALL.	42 43
	FILE_IWRITE_AT_ALL IS a no	IDIOCKING VERSION OF WEI_FILE_WEITE_AT_ALL.	43 44
14.4.3 D	ata Access with Individual Fil	le Pointers	45
			46
	_	er per process per file handle. The current value ffset in the data access routines described in this	47
or this pol	inter implicitly specifies the o	moet in the data access fourmes described in this	48

```
1
     section. These routines only use and update the individual file pointers maintained by MPI.
\mathbf{2}
     The shared file pointer is not used nor updated.
3
          The individual file pointer routines have the same semantics as the data access with
4
     explicit offset routines described in Section 14.4.2, with the following modification:
5
        • the offset is defined to be the current value of the MPI-maintained individual file
6
           pointer.
7
8
     After an individual file pointer operation is initiated, the individual file pointer is updated
9
     to point to the next etype after the last one that will be accessed. The file pointer is updated
10
     relative to the current view of the file.
11
         If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous
12
     to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.
13
14
15
     MPI_FILE_READ(fh, buf, count, datatype, status)
16
       INOUT
                 fh
                                             file handle (handle)
17
       OUT
                 buf
                                             initial address of buffer (choice)
18
19
       IN
                 count
                                             number of elements in buffer (integer)
20
       IN
                                             datatype of each buffer element (handle)
                 datatype
21
       OUT
                 status
                                             status object (status)
22
23
^{24}
     C binding
     int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
25
26
                    MPI_Status *status)
27
     int MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,
28
                    MPI_Datatype datatype, MPI_Status *status)
29
30
     Fortran 2008 binding
     MPI_File_read(fh, buf, count, datatype, status, ierror)
31
32
          TYPE(MPI_File), INTENT(IN) :: fh
33
          TYPE(*), DIMENSION(..) :: buf
34
          INTEGER, INTENT(IN) :: count
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
36
          TYPE(MPI_Status) :: status
37
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_File_read(fh, buf, count, datatype, status, ierror) !(_c)
39
          TYPE(MPI_File), INTENT(IN) :: fh
40
          TYPE(*), DIMENSION(..) :: buf
41
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
          TYPE(MPI_Status) :: status
44
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     Fortran binding
47
     MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
48
          INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

<type> BUF(*)</type>	1
MPI_FILE_READ reads a file using the individual file pointer.	$\frac{2}{3}$
<b>Example 14.2</b> The following Fortran code fragment is an example of reading a file until the end of file is reached:	4 5 6
Read a preexisting input file until all data has been read. Call routine "process_input" if all requested data is read. The Fortran 90 "exit" statement exits the loop.	7 8 9 10
<pre>integer bufsize, numread, totprocessed, status(MPI_STATUS_SIZE) parameter (bufsize=100) real localbuffer(bufsize) integer(kind=MPI_OFFSET_KIND) zero</pre>	11 12 13 14
zero = 0	15 16 17
<pre>call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &amp;</pre>	18 19 20
MPI_INFO_NULL, ierr) totprocessed = 0 do	21 22 23
call MPI_FILE_READ(myfh, localbuffer, bufsize, MPI_REAL, & status, ierr)	24 25
<pre>call MPI_GET_COUNT(status, MPI_REAL, numread, ierr) call process_input(localbuffer, numread)</pre>	26 27 28
<pre>totprocessed = totprocessed + numread if (numread &lt; bufsize) exit end do</pre>	28 29 30
write(6, 1001) numread, bufsize, totprocessed	31 32 33
1001 format("No more data: read", I3, "and expected", I3, & "Processed total of", I6, "before terminating job.")	34 35
call MPI_FILE_CLOSE(myfh, ierr)	36 37

```
1
     MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
                buf
       OUT
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
       IN
                datatype
                                            datatype of each buffer element (handle)
7
       OUT
                status
                                            status object (status)
8
9
10
     C binding
     int MPI_File_read_all(MPI_File fh, void *buf, int count,
11
12
                    MPI_Datatype datatype, MPI_Status *status)
13
     int MPI_File_read_all_c(MPI_File fh, void *buf, MPI_Count count,
14
                    MPI_Datatype datatype, MPI_Status *status)
15
16
     Fortran 2008 binding
17
     MPI_File_read_all(fh, buf, count, datatype, status, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         TYPE(*), DIMENSION(..) :: buf
         INTEGER, INTENT(IN) :: count
20
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Status) :: status
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
     MPI_File_read_all(fh, buf, count, datatype, status, ierror) !(_c)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(..) :: buf
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Status) :: status
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
         <type> BUF(*)
36
         MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_FILE	E_WRITE(fh, buf, cour	ıt, datatype, status)	1	
INOUT	fh	file handle (handle)	2 3	
IN	buf	initial address of buffer (choice)	4	
IN	count	number of elements in buffer (integer)	5	
IN	datatype	datatype of each buffer element (handle)	6	
			7	
OUT	status	status object (status)	8 9	
C binding				
int MPI_File_write(MPI_File fh, const void *buf, int count,				
		datatype, MPI_Status *status)	12	
int MDT			13	
<pre>int MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count,</pre>				
		datatype, mi_Status *status/	15	
Fortran 2008 binding				
MPI_File_write(fh, buf, count, datatype, status, ierror)				
TYPE(MPI_File), INTENT(IN) :: fh TYPE(*), DIMENSION(), INTENT(IN) :: buf				
	GER, INTENT(IN) ::		19 20	
		TENT(IN) :: datatype	21	
	(MPI_Status) :: st		22	
INTE	GER, OPTIONAL, INT	ENT(OUT) :: ierror	23	
<pre>MPI_File_write(fh, buf, count, datatype, status, ierror) !(_c)</pre>				
	(MPI_File), INTENT		25	
		, INTENT(IN) :: buf	26 27	
INTE	GER(KIND=MPI_COUNT	'_KIND), INTENT(IN) :: count	28	
	• •	TENT(IN) :: datatype	29	
TYPE(MPI_Status) :: status				
INTE	GER, UPTIUNAL, INI	ENT(OUT) :: ierror	31	
Fortran binding				
		UNT, DATATYPE, STATUS, IERROR)	33	
		ATYPE, STATUS(MPI_STATUS_SIZE), IERROR	34 35	
<typ< td=""><td>e&gt; BUF(*)</td><td></td><td>36</td></typ<>	e> BUF(*)		36	
MPI_	FILE_WRITE writes a	a file using the individual file pointer.	37	
			38	
			39	
			40	
			41	
			42	
			43 44	
			44	
			46	

```
1
     MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       IN
                buf
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
       IN
                datatype
                                            datatype of each buffer element (handle)
7
       OUT
                status
                                            status object (status)
8
9
10
     C binding
     int MPI_File_write_all(MPI_File fh, const void *buf, int count,
11
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     int MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count,
14
                    MPI_Datatype datatype, MPI_Status *status)
15
16
     Fortran 2008 binding
17
     MPI_File_write_all(fh, buf, count, datatype, status, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
         INTEGER, INTENT(IN) :: count
20
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Status) :: status
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
     MPI_File_write_all(fh, buf, count, datatype, status, ierror) !(_c)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Status) :: status
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
         <type> BUF(*)
36
         MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE inter-
37
     face.
38
39
40
41
42
43
44
45
46
47
48
```

```
1
MPI_FILE_IREAD(fh, buf, count, datatype, request)
                                                                                       \mathbf{2}
  INOUT
           fh
                                      file handle (handle)
                                                                                       3
  OUT
           buf
                                      initial address of buffer (choice)
                                                                                       4
  IN
           count
                                      number of elements in buffer (integer)
                                                                                       5
                                                                                       6
  IN
           datatype
                                      datatype of each buffer element (handle)
                                                                                       7
  OUT
                                      request object (handle)
           request
                                                                                       8
                                                                                       9
C binding
                                                                                       10
int MPI_File_iread(MPI_File fh, void *buf, int count,
                                                                                       11
              MPI_Datatype datatype, MPI_Request *request)
                                                                                       12
                                                                                       13
int MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count,
                                                                                       14
              MPI_Datatype datatype, MPI_Request *request)
                                                                                       15
Fortran 2008 binding
                                                                                       16
MPI_File_iread(fh, buf, count, datatype, request, ierror)
                                                                                       17
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                       18
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                       19
    INTEGER, INTENT(IN) :: count
                                                                                       20
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       21
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       23
                                                                                       24
MPI_File_iread(fh, buf, count, datatype, request, ierror) !(_c)
                                                                                       25
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                       26
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                       27
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                       28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                       29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                       30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                       31
Fortran binding
                                                                                       32
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
                                                                                       33
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
                                                                                       34
    <type> BUF(*)
                                                                                       35
                                                                                       36
    MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.
                                                                                       37
Example 14.3 The following Fortran code fragment illustrates file pointer update seman-
                                                                                       38
tics:
                                                                                       39
                                                                                       40
    Read the first twenty real words in a file into two local
!
                                                                                       41
!
    buffers. Note that when the first MPI_FILE_IREAD returns,
                                                                                       42
    the file pointer has been updated to point to the
!
                                                                                       43
    eleventh real word in the file.
!
                                                                                       44
                                                                                       45
           bufsize, req1, req2
integer
                                                                                       46
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
                                                                                       47
parameter (bufsize=10)
                                                                                       48
```

```
1
     real
                buf1(bufsize), buf2(bufsize)
\mathbf{2}
     integer(kind=MPI_OFFSET_KIND) zero
3
4
     zero = 0
5
     call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
6
                          MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
7
     call MPI_FILE_SET_VIEW(myfh, zero, MPI_REAL, MPI_REAL, 'native', &
8
                              MPI_INFO_NULL, ierr)
9
     call MPI_FILE_IREAD(myfh, buf1, bufsize, MPI_REAL, &
10
                           req1, ierr)
11
     call MPI_FILE_IREAD(myfh, buf2, bufsize, MPI_REAL, &
12
                           req2, ierr)
13
14
     call MPI_WAIT(req1, status1, ierr)
15
     call MPI_WAIT(req2, status2, ierr)
16
17
     call MPI_FILE_CLOSE(myfh, ierr)
18
19
20
     MPI_FILE_IREAD_ALL(fh, buf, count, datatype, request)
21
       INOUT
                fh
                                           file handle (handle)
22
23
       OUT
                buf
                                           initial address of buffer (choice)
24
       IN
                count
                                           number of elements in buffer (integer)
25
26
       IN
                datatype
                                           datatype of each buffer element (handle)
27
       OUT
                request
                                           request object (handle)
28
29
     C binding
30
     int MPI_File_iread_all(MPI_File fh, void *buf, int count,
^{31}
                   MPI_Datatype datatype, MPI_Request *request)
32
33
     int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count,
34
                   MPI_Datatype datatype, MPI_Request *request)
35
     Fortran 2008 binding
36
     MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
37
         TYPE(MPI_File), INTENT(IN) :: fh
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
39
         INTEGER, INTENT(IN) :: count
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_File_iread_all(fh, buf, count, datatype, request, ierror) !(_c)
45
         TYPE(MPI_File), INTENT(IN) :: fh
46
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
47
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
48
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

	YPE(MPI_Request), INTENT(OUT) :: request NTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 2		
INTEGER, OFITONAL, INTENT(OUT) TEITOT				
Fortran binding				
MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)				
	NTEGER FH, COUNT, DATATYPE, REQUEST, IERROR type> BUF(*)	6		
	Cype> BOF(*)	7		
MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.				
		9 10		
MPI_FILE_IWRITE(fh, buf, count, datatype, request)				
INC	UT fh file handle (handle)	12		
IN	buf initial address of buffer (choice)	13		
		14		
IN	count number of elements in buffer (integer)	15		
IN	datatype datatype of each buffer element (handle)	16		
OU	request object (handle)	17 18		
		18		
C binding				
int N	PI_File_iwrite(MPI_File fh, const void *buf, int count,	21		
	MPI_Datatype datatype, MPI_Request *request)	22		
int MPI_File_iwrite_c(MPI_File fh, const void *buf, MPI_Count count,				
	MPI_Datatype datatype, MPI_Request *request)	24		
Fortran 2008 binding				
MPI_File_iwrite(fh, buf, count, datatype, request, ierror)				
TYPE(MPI_File), INTENT(IN) :: fh TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf				
	NTEGER, INTENT(IN) :: count	29 30		
TYPE(MPI_Datatype), INTENT(IN) :: datatype				
TYPE(MPI_Request), INTENT(OUT) :: request				
	NTEGER, OPTIONAL, INTENT(OUT) :: ierror	32 33		
		34		
	<pre>'ile_iwrite(fh, buf, count, datatype, request, ierror) !(_c) 'YPE(MPI_File), INTENT(IN) :: fh</pre>	35		
	YPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	36		
	NTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	37		
	YPE(MPI_Datatype), INTENT(IN) :: datatype	38		
	YPE(MPI_Request), INTENT(OUT) :: request	39		
	NTEGER, OPTIONAL, INTENT(OUT) :: ierror	40		
Fort	an hinding	41		
	an binding ILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)	42		
	NTEGER FH, COUNT, DATATYPE, REQUEST, IERROR	43 44		
	type> BUF(*)	44 45		
		46		
ľ	IPI_FILE_IWRITE is a nonblocking version of MPI_FILE_WRITE.	47		
		48		

```
1
     MPI_FILE_IWRITE_ALL(fh, buf, count, datatype, request)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
       IN
                buf
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
       IN
                datatype
                                            datatype of each buffer element (handle)
7
       OUT
                request
                                            request object (handle)
8
9
10
     C binding
     int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,
11
                    MPI_Datatype datatype, MPI_Request *request)
12
13
     int MPI_File_iwrite_all_c(MPI_File fh, const void *buf, MPI_Count count,
14
                    MPI_Datatype datatype, MPI_Request *request)
15
16
     Fortran 2008 binding
17
     MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
^{18}
         TYPE(MPI_File), INTENT(IN) :: fh
19
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
20
         INTEGER, INTENT(IN) :: count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         TYPE(MPI_Request), INTENT(OUT) :: request
22
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror) !(_c)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
34
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
          <type> BUF(*)
36
         MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.
37
38
39
     MPI_FILE_SEEK(fh, offset, whence)
40
       INOUT
                fh
                                            file handle (handle)
41
42
       IN
                offset
                                            file offset (integer)
43
       IN
                whence
                                            update mode (state)
44
45
     C binding
46
     int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
47
48
```

Fortran 2008 binding							
MPI_File_seek(fh, offset, whence, ierror)							
TYPE(MPI_File), INTENT(IN) :: fh							
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset							
INTEGER, INTENT(IN) :: whenc		5					
INTEGER, OPTIONAL, INTENT(OU	T) :: ierror	6 7					
Fortran binding MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR) INTEGER FH, WHENCE, IERROR							
					INTEGER(KIND=MPI_OFFSET_KIND	) OFFSET	10 11
					MPL FILE SEEK undates the indiv	vidual file pointer according to whence, which has the	12
MPI_FILE_SEEK updates the individual file pointer according to whence, which has the following possible values:							
following possible values.		14					
• MPI_SEEK_SET: the pointer is set	t to offset	15					
• MDI SEEK CUP, the pointer is get to the surrent pointer position plus effect							
• MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset							
• MPI_SEEK_END: the pointer is se	t to the end of file plus offset	18					
		19					
с ,	allows seeking backwards. It is erroneous to seek to	20					
a negative position in the view.		21					
		22 23					
MPI_FILE_GET_POSITION(fh, offset)							
IN fh	file handle (handle)	24 25					
		26					
OUT offset	offset of individual pointer (integer)	27					
		28					
C binding		29					
<pre>int MPI_File_get_position(MPI_Fi</pre>	le ih, MP1_Uiiset *oiiset)	30					
Fortran 2008 binding		31					
<pre>MPI_File_get_position(fh, offset</pre>	, ierror)	32					
TYPE(MPI_File), INTENT(IN) :	: fh	33					
INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset							
INTEGER, OPTIONAL, INTENT(OU	T) :: ierror	35					
Fortran binding							
MPI_FILE_GET_POSITION(FH, OFFSET	TERROR)	37					
INTEGER FH, IERROR							
INTEGER(KIND=MPI_OFFSET_KIND	) OFFSET	39					
		40 41					
MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file pointer in etype units relative to the current view.							
		42					

Advice to users. The offset can be used in a future call to MPI\_FILE\_SEEK using whence = MPI\_SEEK\_SET to return to the current position. To set the displacement to the current file pointer position, first convert offset into an absolute byte position using MPI\_FILE\_GET\_BYTE\_OFFSET, then call MPI\_FILE\_SET\_VIEW with the resulting displacement. (End of advice to users.) 48

```
1
      MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)
2
        IN
                  fh
                                                file handle (handle)
3
        IN
                  offset
                                                offset (integer)
4
5
        OUT
                  disp
                                                absolute byte position of offset (integer)
6
\overline{7}
      C binding
8
      int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
9
                      MPI_Offset *disp)
10
      Fortran 2008 binding
11
      MPI_File_get_byte_offset(fh, offset, disp, ierror)
12
          TYPE(MPI_File), INTENT(IN) :: fh
13
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
14
          INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
15
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
      Fortran binding
18
      MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
19
          INTEGER FH, IERROR
20
          INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
21
          MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte
22
      position. The absolute byte position (from the beginning of the file) of offset relative to the
23
      current view of fh is returned in disp.
^{24}
25
26
              Data Access with Shared File Pointers
      14.4.4
27
      MPI maintains exactly one shared file pointer per collective MPI_FILE_OPEN (shared among
28
      processes in the communicator group). The current value of this pointer implicitly specifies
29
      the offset in the data access routines described in this section. These routines only use and
30
      update the shared file pointer maintained by MPI. The individual file pointers are not used
^{31}
      nor updated.
32
          The shared file pointer routines have the same semantics as the data access with explicit
33
      offset routines described in Section 14.4.2, with the following modifications:
34
35
         • the offset is defined to be the current value of the MPI-maintained shared file pointer,
36
         • the effect of multiple calls to shared file pointer routines is defined to behave as if the
37
           calls were serialized, and
38
39
         • the use of shared file pointer routines is erroneous unless all processes use the same
40
           file view.
41
42
      For the noncollective shared file pointer routines, the serialization ordering is not determin-
43
      istic. The user needs to use other synchronization means to enforce a specific order.
44
          After a shared file pointer operation is initiated, the shared file pointer is updated to
45
      point to the next etype after the last one that will be accessed. The file pointer is updated
46
      relative to the current view of the file.
47
48
```

Noncollec	tive Operations		1
			2
			3
MPI_FILE	E_READ_SHARED(fh, buf, co	unt, datatype, status)	4 5
INOUT	fh	file handle (handle)	6
OUT	buf	initial address of buffer (choice)	7
IN	count	number of elements in buffer (integer)	8
			9
IN	datatype	datatype of each buffer element (handle)	10
OUT	status	status object (status)	11
			12
C bindir	0		13 14
int MPI_		e fh, void *buf, int count,	15
	MPI_Datatype dataty	rpe, MPI_Status *status)	16
int MPI_		ile fh, void *buf, MPI_Count count,	17
	MPI_Datatype dataty	rpe, MPI_Status *status)	18
Fortran	2008 binding		19
MPI_File	_read_shared(fh, buf, co	unt, datatype, status, ierror)	20
	(MPI_File), INTENT(IN) :		21
	(*), DIMENSION() :: bu		22 23
	GER, INTENT(IN) :: count		24
	:(MPI_Datatype), INTENT(I :(MPI_Status) :: status	N) :: datatype	25
	GER, OPTIONAL, INTENT(OU	T) :: ierror	26
			27
		unt, datatype, status, ierror) !(_c)	28
	:(MPI_File), INTENT(IN) : :(*), DIMENSION() :: bu		29
	GER(KIND=MPI_COUNT_KIND)		30 31
	(MPI_Datatype), INTENT(I		31
	(MPI_Status) :: status		33
INTE	GER, OPTIONAL, INTENT(OU	T) :: ierror	34
Fortran	binding		35
	_	UNT, DATATYPE, STATUS, IERROR)	36
		STATUS(MPI_STATUS_SIZE), IERROR	37
	e> BUF(*)		38
MPI	FILE READ SHARED reads	a file using the shared file pointer.	39 40
····· 1_		a me asing the bilated me pointer.	40 41

```
1
     MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)
\mathbf{2}
       INOUT
                fh
                                            file handle (handle)
3
                buf
       IN
                                            initial address of buffer (choice)
4
5
       IN
                                            number of elements in buffer (integer)
                count
6
       IN
                datatype
                                            datatype of each buffer element (handle)
7
       OUT
                status
                                            status object (status)
8
9
10
     C binding
11
     int MPI_File_write_shared(MPI_File fh, const void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)
12
13
     int MPI_File_write_shared_c(MPI_File fh, const void *buf, MPI_Count count,
14
                    MPI_Datatype datatype, MPI_Status *status)
15
16
     Fortran 2008 binding
17
     MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
20
         INTEGER, INTENT(IN) :: count
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Status) :: status
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
     MPI_File_write_shared(fh, buf, count, datatype, status, ierror) !(_c)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         TYPE(MPI_Status) :: status
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
         <type> BUF(*)
36
         MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.
37
38
39
40
41
42
43
44
45
46
47
```

MPI_F	ILE_IREAD_SHARED(f	fh, buf, count, datatype, request)	1
INO	JT fh	file handle (handle)	2
OUT	buf	initial address of buffer (choice)	$\frac{3}{4}$
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
OUT		request object (handle)	7
001	request	request object (nanue)	8 9
C bin	ding		10
		d(MPI_File fh, void *buf, int count,	11
	MPI_Datatyp	e datatype, MPI_Request *request)	12
int M	PI_File_iread_shared	d_c(MPI_File fh, void *buf, MPI_Count count,	13
	MPI_Datatyp	e datatype, MPI_Request *request)	14 15
Fortr	an 2008 binding		16
	U	, buf, count, datatype, request, ierror)	17
	YPE(MPI_File), INTEN		18
		.), ASYNCHRONOUS :: buf	19
	NTEGER, INTENT(IN) : VPF(MPI Datatype) ]	:: count INTENT(IN) :: datatype	20 21
		NTENT(OUT) :: request	22
	-	NTENT(OUT) :: ierror	23
мрт ғ	ile iread shared(fh	, buf, count, datatype, request, ierror) !(_c)	24
	YPE(MPI_File), INTEN		25
	-	.), ASYNCHRONOUS :: buf	26 27
		NT_KIND), INTENT(IN) :: count	28
		INTENT(IN) :: datatype	29
	-	NTENT(OUT) :: request NTENT(OUT) :: ierror	30
		MILMI(UUI) IEIIUI	31
	an binding		32 33
		, BUF, COUNT, DATATYPE, REQUEST, IERROR) ATATYPE, REQUEST, IERROR	34
	type> BUF(*)		35
		RED is a nonblocking version of MPI_FILE_READ_SHARED.	36
IV	FI_FILL_INLAD_SHAN	LED is a holiolocking version of WFI_HEL_NEAD_SHARED.	37
			38 39
			40
			41
			42
			43
			44
			45

```
1
     MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)
2
       INOUT
                 fh
                                             file handle (handle)
3
       IN
                 buf
                                             initial address of buffer (choice)
4
5
       IN
                                             number of elements in buffer (integer)
                 count
6
       IN
                 datatype
                                             datatype of each buffer element (handle)
7
       OUT
                 request
                                             request object (handle)
8
9
10
     C binding
     int MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,
11
                    MPI_Datatype datatype, MPI_Request *request)
12
13
     int MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count,
14
                    MPI_Datatype datatype, MPI_Request *request)
15
16
     Fortran 2008 binding
17
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
18
          TYPE(MPI_File), INTENT(IN) :: fh
19
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
          INTEGER, INTENT(IN) :: count
20
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
          TYPE(MPI_Request), INTENT(OUT) :: request
22
23
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) !(_c)
25
          TYPE(MPI_File), INTENT(IN) :: fh
26
          TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
27
          INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
28
          TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
          TYPE(MPI_Request), INTENT(OUT) :: request
30
          INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
34
          INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
35
          <type> BUF(*)
36
          MPI_FILE_IWRITE_SHARED is a nonblocking version of the
37
     MPI_FILE_WRITE_SHARED interface.
38
39
     Collective Operations
40
41
     The semantics of a collective access using a shared file pointer is that the accesses to the
42
     file will be in the order determined by the ranks of the processes within the group. For each
43
     process, the location in the file at which data is accessed is the position at which the shared
44
     file pointer would be after all processes whose ranks within the group less than that of this
45
     process had accessed their data. In addition, in order to prevent subsequent shared offset
46
     accesses by the same processes from interfering with this collective access, the call might
47
     return only after all the processes within the group have initiated their accesses. When the
48
```

call returns, the shared file pointer points to the next etype accessible, according to the file view used by all processes, after the last etype requested.

Advice to users. There may be some programs in which all processes in the group need to access the file using the shared file pointer, but the program may not *require* that data be accessed in order of process rank. In such programs, using the shared ordered routines (e.g., MPI\_FILE\_WRITE\_ORDERED rather than MPI\_FILE\_WRITE\_SHARED) may enable an implementation to optimize access, improving performance. (*End of advice to users.*)

Advice to implementors. Accesses to the data requested by all processes do not have to be serialized. Once all processes have issued their requests, locations within the file for all accesses can be computed, and accesses can proceed independently from each other, possibly in parallel. (*End of advice to implementors.*)

MPI_FILE	_READ_ORDERED(fh, buf, cou	unt, datatype, status)	17
INOUT	fh	file handle (handle)	18 19
OUT	buf	initial address of buffer (choice)	20
IN	count	number of elements in buffer (integer)	21
			22
IN	datatype	datatype of each buffer element (handle)	23
OUT	status	status object (status)	24
			25
C bindin	g		26
int MPI_H	File_read_ordered(MPI_File	e fh, void *buf, int count,	27
	MPI_Datatype datatype	e, MPI_Status *status)	28
	Cile meed endered c(MDT Ei	the share which MDT Count count	29
int MPI_F		ile fh, void *buf, MPI_Count count,	30
	MFI_Datatype datatyp	e, MPI_Status *status)	31
Fortran 2	2008 binding		32 33
		nt, datatype, status, ierror)	34
	(MPI_File), INTENT(IN) ::	fh	35
	(*), DIMENSION() :: buf		36
	ER, INTENT(IN) :: count		37
	(MPI_Datatype), INTENT(IN)	) :: datatype	38
	(MPI_Status) :: status		39
INTEC	GER, OPTIONAL, INTENT(OUT)	) :: lerror	40
MPI_File_	_read_ordered(fh, buf, cou	<pre>int, datatype, status, ierror) !(_c)</pre>	41
TYPE	(MPI_File), INTENT(IN) ::	fh	42
TYPE	(*), DIMENSION() :: buf		43
	GER(KIND=MPI_COUNT_KIND),		44
TYPE	(MPI_Datatype), INTENT(IN)	) :: datatype	45
	(MPI_Status) :: status		46
INTEC	ER, OPTIONAL, INTENT(OUT)	) :: ierror	47
			48

 $\mathbf{5}$ 

 $\overline{7}$ 

686	
000	

```
1
     Fortran binding
\mathbf{2}
     MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
3
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
4
         <type> BUF(*)
5
         MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED
6
     interface.
7
8
9
     MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)
10
       INOUT
                fh
                                           file handle (handle)
11
       IN
                buf
                                           initial address of buffer (choice)
12
13
       IN
                count
                                           number of elements in buffer (integer)
14
       IN
                datatype
                                           datatype of each buffer element (handle)
15
16
       OUT
                status
                                           status object (status)
17
18
     C binding
19
     int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
20
                    MPI_Datatype datatype, MPI_Status *status)
21
     int MPI_File_write_ordered_c(MPI_File fh, const void *buf, MPI_Count count,
22
                    MPI_Datatype datatype, MPI_Status *status)
23
^{24}
     Fortran 2008 binding
25
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
26
         TYPE(MPI_File), INTENT(IN) :: fh
27
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
28
         INTEGER, INTENT(IN) :: count
29
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
         TYPE(MPI_Status) :: status
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_File_write_ordered(fh, buf, count, datatype, status, ierror) !(_c)
33
         TYPE(MPI_File), INTENT(IN) :: fh
34
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
35
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         TYPE(MPI_Status) :: status
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     Fortran binding
41
     MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
42
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
43
         <type> BUF(*)
44
         MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED
45
     interface.
46
47
48
```

Seek			1
If MPI_MO	DE_SEQUENTIAL mode was a	specified when the file was opened, it is erroneous	2 3
	3	$PI_FILE_SEEK_SHARED$ and	4
MPI_FILE	_GET_POSITION_SHARED).		5
			6
MPI_FILE	_SEEK_SHARED(fh, offset, wh	nence)	7
INOUT	fh	file handle (handle)	8 9
IN	offset	file offset (integer)	10
IN	whence		11
IIN	whence	update mode (state)	12
C binding	σ		13
	-	fh, MPI_Offset offset, int whence)	14
		, <u>-</u> ,	15 16
	2008 binding _seek_shared(fh, offset,	whence jerror)	10
	(MPI_File), INTENT(IN) ::		18
	GER(KIND=MPI_OFFSET_KIND)		19
	GER, INTENT(IN) :: whence		20
INTEG	GER, OPTIONAL, INTENT(OUT	) :: ierror	21
Fortran b	binding		22 23
MPI_FILE_	_SEEK_SHARED(FH, OFFSET,	WHENCE, IERROR)	23
	ER FH, WHENCE, IERROR		25
INTEG	GER(KIND=MPI_OFFSET_KIND)	OFFSET	26
MPI_I	FILE_SEEK_SHARED updates	s the shared file pointer according to whence, which	27
has the fol	lowing possible values:		28
• MPI_	SEEK_SET: the pointer is set	to offset	29 30
• MPI_	SEEK_CUR: the pointer is set	to the current pointer position plus offset	31 32
<ul> <li>MDI</li> </ul>	SEEK_END: the pointer is set	to the end of file plus offset	33
● IVIFI_	SEEK_END. the pointer is set	to the end of me plus onset	34
		ctive; all the processes in the communicator group	35
		all MPI_FILE_SEEK_SHARED with the same values	36
	nd whence.	llows seeking backwards. It is erroneous to seek to	37
	position in the view.	mows seeking backwards. It is enoneous to seek to	38 39
a nogativo			40
			41
	_GET_POSITION_SHARED(fl	,	42
IN	fh	file handle (handle)	43
OUT	offset	offset of shared pointer (integer)	44
			45 46
C binding	-		40
int MPI_F	'iie_get_position_shared(	MPI_File fh, MPI_Offset *offset)	48

1	Fortran 2008 binding
2	MPI_File_get_position_shared(fh, offset, ierror)
3	TYPE(MPI_File), INTENT(IN) :: fh
4	INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
5	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6	
7	Fortran binding
8	MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
9	INTEGER FH, IERROR
10	INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
11	MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the
12	
13	shared file pointer in etype units relative to the current view.
14	Advice to users. The offset can be used in a future call to MPI_FILE_SEEK_SHARED
15	
	using whence = MPI_SEEK_SET to return to the current position. To set the displace-
16	ment to the current file pointer position, first convert offset into an absolute byte
17	position using MPI_FILE_GET_BYTE_OFFSET, then call MPI_FILE_SET_VIEW with
18	the resulting displacement. (End of advice to users.)
19	
20	14.4.5 Split Collective Data Access Routines
21	MPI provides a postnicted form of "pophlacking collective" I/O operations for all data as
22	MPI provides a restricted form of "nonblocking collective" I/O operations for all data ac-
23	cesses using split collective data access routines. These routines are referred to as "split"
24	collective routines because a single collective operation is split in two: a begin routine and
25	an end routine. The begin routine begins the operation, much like a nonblocking data access
26	(e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching
27	test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must
28	not use the buffer passed to a begin routine while the routine is outstanding; the operation
29	must be completed with an end routine before it is safe to free buffers, etc.
30	Split collective data access operations on a file handle fh are subject to the semantic
31	rules given below.
32	
33	• On any MPI process, each file handle may have at most one active split collective
34	operation at any time.
35	
	• Begin calls are collective over the group of processes that participated in the collective
36 27	open and follow the ordering rules for collective calls.
37	
38	• End calls are collective over the group of processes that participated in the collective
39	open and follow the ordering rules for collective calls. Each end call matches the
40	preceding begin call for the same collective operation. When an "end" call is made,
41	exactly one unmatched "begin" call for the same operation must precede it.
42	• An implementation is free to implement any split collective data access routine using
43	• An implementation is nee to implement any spirt concerve data access routine using the corresponding blocking collective routine when either the begin call (e.g.,
44	MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is
45	
46	issued. The begin and end calls are provided to allow the user and MPI implementation
47	to optimize the collective operation.
48	

According to the definitions in Section 2.4.2, the begin procedures are incomplete. They are also non-local procedures because they may or may not return before they are called in all MPI processes of the process group.

Advice to users. This is one of the exceptions in which incomplete procedures are non-local and therefore blocking. (End of advice to users.)

- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI\_FILE\_READ\_ALL on one process does not match an MPI\_FILE\_READ\_ALL\_BEGIN/ MPI\_FILE\_READ\_ALL\_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in "A Problem with Code Movements and Register Optimization," Section 19.1.17, but not all of the problems, such as those described in Sections 19.1.12, 19.1.13, and 19.1.16.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

```
MPI_File_read_all_begin(fh, ...);
...
MPI_File_read_all(fh, ...);
...
MPI_File_read_all_end(fh, ...);
```

is erroneous.

• In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for MPI\_FILE\_READ\_ALL\_BEGIN and MPI\_FILE\_READ\_ALL\_END are equivalent to the arguments for MPI\_FILE\_READ\_ALL). The begin routine (e.g., MPI\_FILE\_READ\_ALL\_BEGIN) begins a split collective operation that, when completed with the matching end routine (i.e., MPI\_FILE\_READ\_ALL\_END) produces the result as defined for the equivalent collective routine (i.e., MPI\_FILE\_READ\_ALL).

For the purpose of consistency semantics (Section 14.6.1), a matched pair of split collective data access operations (e.g., MPI\_FILE\_READ\_ALL\_BEGIN and MPI\_FILE\_READ\_ALL\_END) compose a single data access.

 $^{24}$ 

 $^{31}$ 

```
1
     MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)
\mathbf{2}
       IN
                fh
                                            file handle (handle)
3
                offset
       IN
                                            file offset (integer)
4
5
       OUT
                buf
                                            initial address of buffer (choice)
6
       IN
                count
                                            number of elements in buffer (integer)
7
       IN
                datatype
                                            datatype of each buffer element (handle)
8
9
10
     C binding
     int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
11
                    int count, MPI_Datatype datatype)
12
13
     int MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf,
14
                    MPI_Count count, MPI_Datatype datatype)
15
16
     Fortran 2008 binding
17
     MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
18
         TYPE(MPI_File), INTENT(IN) :: fh
19
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
20
21
         INTEGER, INTENT(IN) :: count
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror) !(_c)
25
         TYPE(MPI_File), INTENT(IN) :: fh
26
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
27
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
28
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
29
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     Fortran binding
33
     MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
34
          INTEGER FH, COUNT, DATATYPE, IERROR
35
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
36
         <type> BUF(*)
37
38
39
     MPI_FILE_READ_AT_ALL_END(fh, buf, status)
40
                fh
       IN
                                            file handle (handle)
41
42
       OUT
                buf
                                            initial address of buffer (choice)
43
       OUT
                status
                                            status object (status)
44
45
     C binding
46
     int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
47
48
```

	2008 binding		1
		l(fh, buf, status, ierror)	2
	(MPI_File), INTE		3 4
	(*), DIMENSION(. (MPI_Status) ::	.), ASYNCHRONOUS :: buf	5
		INTENT(OUT) :: ierror	6
			7
Fortran	0	)(FH, BUF, STATUS, IERROR)	8
		IPI_STATUS_SIZE), IERROR	9
	e> BUF(*)		10 11
			12
			13
MPI_FILE	_WRITE_AT_ALL_	BEGIN(fh, offset, buf, count, datatype)	14
INOUT	fh	file handle (handle)	15
IN	offset	file offset (integer)	16 17
IN	buf	initial address of buffer (choice)	18
IN	count	number of elements in buffer (integer)	19
IN	datatype	datatype of each buffer element (handle)	20 21
			22
C bindin	0		23
int MPI_		.l_begin(MPI_File fh, MPI_Offset offset,	24
	const void	*buf, int count, MPI_Datatype datatype)	25
int MPI_		l_begin_c(MPI_File fh, MPI_Offset offset,	26
	const void	<pre>*buf, MPI_Count count, MPI_Datatype datatype)</pre>	27 28
Fortran 2	2008 binding		29
		egin(fh, offset, buf, count, datatype, ierror)	30
	(MPI_File), INTE		31
		SET_KIND), INTENT(IN) :: offset	32
	GER, INTENT(IN)	.), INTENT(IN), ASYNCHRONOUS :: buf	33
		INTENT(IN) :: datatype	34 35
	• -	INTENT(OUT) :: ierror	36
MDT Eile	unite et ell he	win (the offerst buf count determs is man) I (c)	37
	(MPI_File), INTE	<pre>egin(fh, offset, buf, count, datatype, ierror) !(_c) NT(IN) :: fh</pre>	38
	-	SET_KIND), INTENT(IN) :: offset	39
		.), INTENT(IN), ASYNCHRONOUS :: buf	40
		<pre>JNT_KIND), INTENT(IN) :: count</pre>	41 42
		INTENT(IN) :: datatype	42
INTE	GER, OPTIONAL, I	INTENT(OUT) :: ierror	44
Fortran	binding		45
		GIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)	46
		DATATYPE, IERROR	47
TN.LE	GFK(KIND=WLT_OFF	SET_KIND) OFFSET	48

```
1
         <type> BUF(*)
\mathbf{2}
3
4
     MPI_FILE_WRITE_AT_ALL_END(fh, buf, status)
5
       INOUT
                fh
                                            file handle (handle)
6
       IN
7
                 buf
                                            initial address of buffer (choice)
8
       OUT
                                            status object (status)
                status
9
10
     C binding
11
     int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
12
                    MPI_Status *status)
13
14
     Fortran 2008 binding
15
     MPI_File_write_at_all_end(fh, buf, status, ierror)
16
         TYPE(MPI_File), INTENT(IN) :: fh
17
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
18
         TYPE(MPI_Status) :: status
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     Fortran binding
21
     MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
22
          INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
23
          <type> BUF(*)
24
25
26
     MPI_FILE_READ_ALL_BEGIN(fh, buf, count, datatype)
27
28
       INOUT
                fh
                                            file handle (handle)
29
       OUT
                buf
                                            initial address of buffer (choice)
30
       IN
                count
                                            number of elements in buffer (integer)
^{31}
32
       IN
                datatype
                                            datatype of each buffer element (handle)
33
34
     C binding
35
     int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
36
                    MPI_Datatype datatype)
37
     int MPI_File_read_all_begin_c(MPI_File fh, void *buf, MPI_Count count,
38
                    MPI_Datatype datatype)
39
40
     Fortran 2008 binding
41
     MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
42
         TYPE(MPI_File), INTENT(IN) :: fh
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
44
         INTEGER, INTENT(IN) :: count
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
     MPI_File_read_all_begin(fh, buf, count, datatype, ierror) !(_c)
48
```

TYPE	(MPI_File), INTENT(IN) ::	fh	1
	(*), DIMENSION(), ASYNC		2
	GER(KIND=MPI_COUNT_KIND),		3
	(MPI_Datatype), INTENT(IN		4 5
INTEC	ER, OPTIONAL, INTENT(OUT	) :: lerror	6
Fortran l	binding		7
MPI_FILE	READ_ALL_BEGIN(FH, BUF,	COUNT, DATATYPE, IERROR)	8
	ER FH, COUNT, DATATYPE,	IERROR	9
<type< td=""><td>&gt; BUF(*)</td><td></td><td>10</td></type<>	> BUF(*)		10
			11
		、 、	12
MPI_FILE	_READ_ALL_END(fh, buf, stat	tus)	13
INOUT	fh	file handle (handle)	14
OUT	buf	initial address of buffer (choice)	15 16
OUT	status	status object (status)	10
001	Status	Startas object (Startas)	18
C bindin	σ		19
	0	e fh, void *buf, MPI_Status *status)	20
			21
	2008 binding		22
	_read_all_end(fh, buf, st (MPI_File), INTENT(IN) ::		23
	(*), DIMENSION(), ASYNC		24
	(MPI_Status) :: status		25
	ER, OPTIONAL, INTENT(OUT	) :: ierror	26 27
Dantara I	· · · · · · · · · · · ·		27
Fortran l	_READ_ALL_END(FH, BUF, ST		29
	ER FH, STATUS(MPI_STATUS)	-	30
	e> BUF(*)		31
51			32
			33
MPI_FILE	WRITE_ALL_BEGIN(fh, buf,	count, datatype)	34
INOUT	fh	file handle (handle)	35
			36
IN	buf	initial address of buffer (choice)	37 38
IN	count	number of elements in buffer (integer)	39
IN	datatype	datatype of each buffer element (handle)	40
			41
C bindin	g		42
int MPI_H	0	File fh, const void *buf, int count,	43
	MPI_Datatype datatyp	e)	44
int MPI_H	File_write_all_begin_c(MP)	I_File fh, const void *buf,	45
	MPI_Count count, MPI		46
			47 48
			40

```
Fortran 2008 binding
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
   TYPE(MPI_File), INTENT(IN) :: fh
   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
   INTEGER, INTENT(IN) :: count
   TYPE(MPI_Datatype), INTENT(IN) :: datatype
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
   TYPE(MPI_File) INTENT(IN) :: fb
```

```
5
6
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     MPI_File_write_all_begin(fh, buf, count, datatype, ierror) !(_c)
9
         TYPE(MPI_File), INTENT(IN) :: fh
10
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
11
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     Fortran binding
16
     MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
17
         INTEGER FH, COUNT, DATATYPE, IERROR
18
         <type> BUF(*)
19
20
21
     MPI_FILE_WRITE_ALL_END(fh, buf, status)
22
       INOUT
                fh
                                           file handle (handle)
23
^{24}
       IN
                buf
                                           initial address of buffer (choice)
25
                                           status object (status)
       OUT
                status
26
27
     C binding
28
     int MPI_File_write_all_end(MPI_File fh, const void *buf,
29
                   MPI_Status *status)
30
31
     Fortran 2008 binding
32
     MPI_File_write_all_end(fh, buf, status, ierror)
33
         TYPE(MPI_File), INTENT(IN) :: fh
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
35
         TYPE(MPI_Status) :: status
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     Fortran binding
38
     MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
39
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
40
         <type> BUF(*)
41
42
43
44
45
46
```

 $\mathbf{2}$ 

3

4

MPI_FILE	_READ_ORDER	ED_BEGIN(fh, buf, count, datatype)	1
INOUT	fh	file handle (handle)	2 3
OUT	buf	initial address of buffer (choice)	4
IN	count	number of elements in buffer (integer)	5
IN	datatype	datatype of each buffer element (handle)	6
			7 8
C bindin	g		9
int MPI_H		ered_begin(MPI_File fh, void *buf, int count,	10
	MPI_Data	type datatype)	11
int MPI_H		ered_begin_c(MPI_File fh, void *buf, MPI_Count count, type datatype)	12 13
Fortran 2	2008 binding		14 15
MPI_File_	_read_ordered_	_begin(fh, buf, count, datatype, ierror)	16
		NTENT(IN) :: fh	17
	(*), DIMENSION GER, INTENT(IN	N(), ASYNCHRONOUS :: buf	18
		), INTENT(IN) :: datatype	19 20
	• -	, INTENT(OUT) :: ierror	20 21
MPT File	read ordered	_begin(fh, buf, count, datatype, ierror) !(_c)	22
		VTENT(IN) :: fh	23
		N(), ASYNCHRONOUS :: buf	24
		COUNT_KIND), INTENT(IN) :: count	25 26
	• -	), INTENT(IN) :: datatype	20
INTEC	ER, UPIIUNAL,	, INTENT(OUT) :: ierror	28
Fortran h			29
		_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) , DATATYPE, IERROR	30
	e> BUF(*)	DATATIFE, TEMOR	31 32
-71			33
			34
MPI_FILE	_READ_ORDER	ED_END(fh, buf, status)	35
INOUT	fh	file handle (handle)	36
OUT	buf	initial address of buffer (choice)	37 38
OUT	status	status object (status)	39
			40
C bindin	g		41
int MPI_H	File_read_orde	ered_end(MPI_File fh, void *buf, MPI_Status *status)	42
Fortran 2	2008 binding		43 44
		_end(fh, buf, status, ierror)	45
	-	NTENT(IN) :: fh	46
		V(), ASYNCHRONOUS :: buf	47
1 I PE	(MPI_Status) :	: Status	48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     Fortran binding
3
     MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
4
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
5
         <type> BUF(*)
6
7
8
     MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)
9
10
                                           file handle (handle)
       INOUT
                fh
11
       IN
                buf
                                           initial address of buffer (choice)
12
                                           number of elements in buffer (integer)
       IN
                count
13
14
       IN
                datatype
                                           datatype of each buffer element (handle)
15
16
     C binding
17
     int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
18
                    MPI_Datatype datatype)
19
     int MPI_File_write_ordered_begin_c(MPI_File fh, const void *buf,
20
                    MPI_Count count, MPI_Datatype datatype)
21
22
     Fortran 2008 binding
23
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
24
         TYPE(MPI_File), INTENT(IN) :: fh
25
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
26
         INTEGER, INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
     MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) !(_c)
30
         TYPE(MPI_File), INTENT(IN) :: fh
^{31}
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
32
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
33
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     Fortran binding
37
     MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
38
         INTEGER FH, COUNT, DATATYPE, IERROR
39
         <type> BUF(*)
40
41
42
43
44
45
46
47
48
```

MPI_FILE_WRITE_ORDERED_END(fh, buf, status)			
INOUT	fh	file handle (handle)	
IN	buf	initial address of buffer (choice)	
OUT	status	status object (status)	

### C binding

### Fortran 2008 binding

```
MPI_File_write_ordered_end(fh, buf, status, ierror)
   TYPE(MPI_File), INTENT(IN) :: fh
   TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
   TYPE(MPI_Status) :: status
   INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### Fortran binding

```
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
    <type> BUF(*)
```

# 14.5 File Interoperability

At the most basic level, file interoperability is the ability to read the information previously written to a file—not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 14.5.2) as well as the data conversion functions (Section 14.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 14.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI\_COMM\_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar

 $^{31}$ 

1 operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it  $\mathbf{2}$ is expected that the facility provided maintains the correspondence between absolute byte 3 offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the 4 MPI environment are at byte offset 102 outside the MPI environment). As an example,  $\mathbf{5}$ a simple off-line conversion utility that transfers and converts files between the native file 6 system and the MPI environment would suffice, provided it maintained the offset coherence  $\overline{7}$ mentioned above. In a high-quality implementation of MPI, users will be able to manipulate 8 MPI files using the same or similar tools that the native file system offers for manipulating 9 its files.

10 The remaining aspect of file interoperability, converting between different machine 11representations, is supported by the typing information specified in the etype and filetype. 12This facility allows the information in files to be shared between any two applications, 13regardless of whether they use MPI, and regardless of the machine architectures on which 14they run.

15MPI supports multiple data representations: "native", "internal", and "external32". An 16implementation may support additional data representations. MPI also supports user-17defined data representations (see Section 14.5.3). The "native" and "internal" data repre-18 sentations are implementation dependent, while the "external32" representation is common 19to all MPI implementations and facilitates file interoperability. The data representation is 20specified in the datarep argument to MPI\_FILE\_SET\_VIEW.

- Advice to users. MPI is not guaranteed to retain knowledge of what data representation was used when a file is written. Therefore, to correctly retrieve file data, an MPI application is responsible for specifying the same data representation as was used to create the file. (End of advice to users.)
- 26"native" Data in this representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment.
  - Advice to users. This data representation should only be used in a homogeneous MPI environment, or when the MPI application is capable of performing the datatype conversions itself. (End of advice to users.)
  - Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be typed as MPI\_BYTE to ensure that the message routines do not perform any type conversions on the data. (End of advice to implementors.)
- 39 "internal" This data representation can be used for I/O operations in a homogeneous or 40heterogeneous environment; the implementation will perform type conversions if nec-41 essary. The implementation is free to store data in any format of its choice, with the 42restriction that it will maintain constant extents for all predefined datatypes in any 43 one file. The environment in which the resulting file can be reused is implementation-44defined and must be documented by the implementation. 45
- 46*Rationale.* This data representation allows the implementation to perform I/O47 efficiently in a heterogeneous environment, though with implementation-defined 48 restrictions on how the file can be reused. (*End of rationale.*)

21

22

23

 $^{24}$ 

25

27

28

29

30

31

32

33 34

35

36

37

"external32" This data representation states that read and write operations convert all data from and to the "external32" representation defined in Section 14.5.2. The data conversion rules for communication also apply to these conversions (see Section 3.3.2). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in datatype conversions.

Advice to implementors. When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI\_BYTE. This will avoid possible double datatype conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

### 14.5.1 Datatypes for File Interoperability

If the file data representation is other than "native", care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI\_FILE\_GET\_TYPE\_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

One can logically think of the file as if it were stored in the 37 Advice to users. memory of a file server. The etype and filetype are interpreted as if they were defined 3839 at this file server, by the same sequence of calls used to define them at the calling process. If the data representation is "native", then this logical file server runs on 40 41 the same architecture as the calling process, so that these types define the same data 42layout on the file as they would define in the memory of the calling process. If the 43etype and filetype are portable datatypes, then the data layout defined in the file is the same as would be defined in the calling process memory, up to a scaling factor. 44The routine MPI\_FILE\_GET\_TYPE\_EXTENT can be used to calculate this scaling 4546factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with "internal", "external32", or user 4748 defined data representations. Otherwise, the etype and filetype must be constructed

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7

8

9 10

11

12

13

14

15

16

17 18

19

20

21

22

23 24

25 26

27

28

29

30

 $^{31}$ 

32

33

34

35

1 so that their typemap and extent are the same on any architecture. This can be  $\mathbf{2}$ achieved if they have an explicit upper bound and lower bound (defined using 3 MPI\_TYPE\_CREATE\_RESIZED). This condition must also be fulfilled by any datatype 4 that is used in the construction of the etype and filetype, if this datatype is replicated 5contiguously, either explicitly, by a call to MPI\_TYPE\_CONTIGUOUS, or implicitly, 6 by a blocklength argument that is greater than one. If an etype or filetype is not 7 portable, and has a typemap or extent that is architecture dependent, then the data 8 layout specified by it on a file is implementation dependent. 9 File data representations other than "native" may be different from corresponding 10 data representations in memory. Therefore, for these file data representations, it is 11 important not to use hardwired byte offsets for file positioning, including the initial 12displacement that specifies the view. When a portable datatype (see Section 2.4) is 13 used in a data access operation, any holes in the datatype are scaled to match the data 14representation. However, note that this technique only works when all the processes 15that created the file view build their etypes from the same predefined datatypes. For 16example, if one process uses an etype built from MPI\_INT and another uses an etype 17 built from MPI\_FLOAT, the resulting views may be nonportable because the relative 18 sizes of these types may differ from one data representation to another. (End of advice 19 to users.) 202122 MPI\_FILE\_GET\_TYPE\_EXTENT(fh, datatype, extent) 23 $^{24}$ IN fh file handle (handle) 25IN datatype datatype (handle) 26OUT extent datatype extent (integer) 2728C binding 2930 int MPI\_File\_get\_type\_extent(MPI\_File fh, MPI\_Datatype datatype,  $^{31}$ MPI Aint \*extent) 32 int MPI\_File\_get\_type\_extent\_c(MPI\_File fh, MPI\_Datatype datatype, 33 MPI\_Count \*extent) 3435 Fortran 2008 binding 36 MPI\_File\_get\_type\_extent(fh, datatype, extent, ierror) 37 TYPE(MPI\_File), INTENT(IN) :: fh 38 TYPE(MPI\_Datatype), INTENT(IN) :: datatype INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(OUT) :: extent 39 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI\_File\_get\_type\_extent(fh, datatype, extent, ierror) !(\_c) 42TYPE(MPI\_File), INTENT(IN) :: fh 43 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 44 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(OUT) :: extent 45INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4647Fortran binding 48 MPI\_FILE\_GET\_TYPE\_EXTENT(FH, DATATYPE, EXTENT, IERROR)

### INTEGER FH, DATATYPE, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTENT

Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 14.5.3), MPI uses the dtype\_file\_extent\_fn callback to calculate the extent.

If the datatype extent cannot be represented in extent, it is set to MPI\_UNDEFINED.

Advice to implementors. In the case of user-defined data representations, the extent of a derived datatype can be calculated by first determining the extents of the predefined datatypes in this derived datatype using dtype\_file\_extent\_fn (see Section 14.5.3). (End of advice to implementors.)

### 14.5.2 External Data Representation: "external32"

All MPI implementations are required to support the data representation defined in this section. Support of optional datatypes (e.g., MPI\_INTEGER2) is not required.

All floating point values are in big-endian IEEE format [42] of the appropriate size. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single (binary32)," "Double (binary64)," and "Double Extended (binary128)" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended (binary128)" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +16383, 112 fraction bits, and an encoding analogous to the "Double (binary64)" format. All integral values are in two's complement big-endian format. Big-endian means most significant byte at lowest address byte. For C \_Bool, Fortran LOGICAL, and C++ bool, 0 implies false and nonzero implies true. C float \_Complex, double \_Complex, and long double \_Complex, Fortran COMPLEX and DOUBLE COMPLEX, and other complex types are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [43]. Wide characters (of type MPI\_WCHAR) are in Unicode format [68].

All signed numerals (e.g., MPI\_INT, MPI\_REAL) have the sign bit at the most significant bit. MPI\_COMPLEX and MPI\_DOUBLE\_COMPLEX have the sign bit of the real and imaginary parts at the most significant bit of each part.

According to IEEE specifications [42], the "NaN" (not a number) is system dependent. It should not be interpreted within MPI as anything other than "NaN."

Advice to implementors. The MPI treatment of "NaN" is similar to the approach used in XDR [65]. (End of advice to implementors.)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

Advice to implementors. All bytes of LOGICAL and bool must be checked to determine the value. (End of advice to implementors.)

Advice to users. The type MPI\_PACKED is treated as bytes and is not converted. The user should be aware that MPI\_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

1

 $\mathbf{2}$ 

3

4

5

6

7 8

9

10

11

12 13

14 15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37 38

39

40 41

42

43 44

45

46

Predefined Type	Length
MPI_PACKED	1
MPI_BYTE	1
MPI_CHAR	1
MPI_UNSIGNED_CHAR	1
MPI_SIGNED_CHAR	1
MPI_WCHAR	2
MPI_SHORT	2
MPI_UNSIGNED_SHORT	2
MPI_INT	4
MPI_LONG	4
MPI_UNSIGNED	4
MPI_UNSIGNED_LONG	4
MPI_LONG_LONG_INT	8
MPI_UNSIGNED_LONG_LONG	8
MPI_FLOAT	4
MPI_DOUBLE	8
MPI_LONG_DOUBLE	16
MPI_C_BOOL	1
MPI_INT8_T	1
MPI_INT16_T	2
MPI_INT32_T	4
MPI_INT64_T	8
MPI_UINT8_T	1
MPI_UINT16_T	2
MPI_UINT32_T	4
MPI_UINT64_T	8
MPI_AINT	8
MPI_COUNT	8
MPI_OFFSET	8
MPI_C_COMPLEX	$2^{*}4$
MPI_C_FLOAT_COMPLEX	$2^{*4}$
MPI_C_DOUBLE_COMPLEX	2*8
MPI_C_DOUBLE_COMPLEX	$2^{*}16$
MPI_CHARACTER	1
MPI_LOGICAL	4
MPI_INTEGER	4
MPI_REAL	4
MPI_DOUBLE_PRECISION	8
MPI_COMPLEX	2*4
MPI_DOUBLE_COMPLEX	2*8
MPI_CXX_BOOL	1
MPI_CXX_FLOAT_COMPLEX	2*4
	1
MPI_CXX_DOUBLE_COMPLEX	2*8

Table 14.2: "external32" sizes of predefined datatypes

47

Predefined Type	Length
MPI_INTEGER1	1
MPI_INTEGER2	2
MPI_INTEGER4	4
MPI_INTEGER8	8
MPI_INTEGER16	16
MPI_REAL2	2
MPI_REAL4	4
MPI_REAL8	8
MPI_REAL16	16
MPI_COMPLEX4	$2^{*}2$
MPI_COMPLEX8	2*4
MPI_COMPLEX16	2*8
MPI_COMPLEX32	2*16

Table 14.3:	"external32"	sizes	of opt	ional	datatypes
10010 11.0.	CALCHING 2	01200	or opt	ronu	addady pob

C++ Types	Length
MPI_CXX_BOOL	1
MPI_CXX_FLOAT_COMPLEX	2*4
MPI_CXX_DOUBLE_COMPLEX	$2^{*8}$
MPI_CXX_LONG_DOUBLE_COMPLEX	2*16

Table 14.4: "external32" sizes of C++ datatypes	
---	--

1 2	MPI_TY	PE_CREATE_F90_COMPLEX	ypes returned from MPI_TYPE_CREATE_F90_REAL, K, and MPI_TYPE_CREATE_F90_INTEGER are defined	
3 4	in Sectio	on 19.1.9, page 813.		
5 6 7 8	$\operatorname{int}$	eger, only the least significate sign bit value. This allows	nen converting a larger size integer to a smaller size nt bytes are moved. Care must be taken to preserve s no conversion errors if the data range is within the r. ( <i>End of advice to implementors.</i> )	
9 10 11	Table 14.2, 14.3, and 14.4 specify the sizes of predefined, optional, and C++ datatypes in "external32" format, respectively.			
12 13	14.5.3	User-Defined Data Represer	ntations	
$14 \\ 15$	There ar	e two situations that cannot	be handled by the required representations:	
16	1. a u	user wants to write a file in a	representation unknown to the implementation, and	
17 18	2. a u	ser wants to read a file writte	en in a representation unknown to the implementation.	
19 20 21 22		r-defined data representation stream to do the data repres	as allow the user to insert a third party converter into sentation conversion.	
23 24	MPI_RE	GISTER_DATAREP(datarep, dtype_file_extent_fn,	read_conversion_fn, write_conversion_fn, extra_state)	
25 26	IN	datarep	data representation identifier (string)	
27 28	IN	read_conversion_fn	function invoked to convert from file representation to native representation (function)	
29 30	IN	write_conversion_fn	function invoked to convert from native representation to file representation (function)	
31 32 33	IN	dtype_file_extent_fn	function invoked to get the extent of a datatype as represented in the file (function)	
34	IN	extra_state	extra state	
35 36	C bindi	nσ		
37		_Register_datarep(const	char *datarep.	
38		•	rsion_function *read_conversion_fn,	
39		-	rsion_function *write_conversion_fn,	
40	MPI_Datarep_extent_function *dtype_file_extent_fn,			
41		void *extra_state	)	
42	int MPT	_Register_datarep_c(cons	st char *datarep.	
43	•		rsion_function_c *read_conversion_fn,	
44 45		-	rsion_function_c *write_conversion_fn,	
45 46		-	t_function *dtype_file_extent_fn,	
47		void *extra_state	)	
48				

```
Fortran 2008 binding
                                                                                     1
                                                                                     2
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
              dtype_file_extent_fn, extra_state, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                     4
    PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn,
                                                                                     5
                                                                                     6
               write_conversion_fn
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
                                                                                     7
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     9
                                                                                     10
MPI_Register_datarep_c(datarep, read_conversion_fn, write_conversion_fn,
                                                                                     11
              dtype_file_extent_fn, extra_state, ierror) !(_c)
                                                                                     12
    CHARACTER(LEN=*), INTENT(IN) :: datarep
                                                                                     13
    PROCEDURE(MPI_Datarep_conversion_function_c) :: read_conversion_fn,
                                                                                    14
               write_conversion_fn
                                                                                     15
    PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
                                                                                     16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                     17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                     18
                                                                                     19
Fortran binding
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
                                                                                    20
                                                                                    21
              DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
                                                                                    22
    CHARACTER*(*) DATAREP
                                                                                    23
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
                                                                                    24
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
                                                                                    25
    INTEGER IERROR
                                                                                    26
    The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn
                                                                                    27
with the data representation identifier datarep. datarep can then be used as an argument
                                                                                    28
to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conver-
                                                                                    29
sion functions to convert all data items accessed between file data representation and na-
                                                                                    30
```

tive representation. MPI\_REGISTER\_DATAREP is a local operation and only registers the data representation for the calling MPI process. If datarep is already defined, an error in the error class MPI\_ERR\_DUP\_DATAREP is raised using the default file error handler (see Section 14.7). The length of a data representation string is limited to the value of MPI\_MAX\_DATAREP\_STRING. MPI\_MAX\_DATAREP\_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

```
Extent Callback
```

```
40
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
                                                                                   41
             MPI_Aint *extent, void *extra_state);
                                                                                   42
ABSTRACT INTERFACE
                                                                                   43
 SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
                                                                                   44
              ierror)
                                                                                   45
    TYPE(MPI_Datatype) :: datatype
                                                                                   46
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
                                                                                   47
    INTEGER :: ierror
                                                                                   48
```

 $^{31}$ 

32

33

34

35

36

37 38

1 2 3	SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
4	
5	The function dtype_file_extent_fn must return, in file_extent, the number of bytes re-
6	quired to store datatype in the file representation. The function is passed, in extra_state, the argument that was passed to the MPI PECISTEP DATAPED call. MPI will only call
7	the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call this routine with predefined datatypes employed by the user.
8	this fourne with predemied datatypes employed by the user.
9 10	Rationale. This callback does not have a large count variant because it is anticipated
10	that large counts will not be required to represent the extent output value. (End of
12	rationale.)
13	MPI_Datarep_conversion_function also supports large count types in separate additional
14	MPI procedures in C (suffixed with the " $_c$ ") and multiple abstract interfaces in Fortran
15	when using USE mpi_f08.
16	If the extent cannot be represented in extent, the callback function shall set extent to
17	MPI_UNDEFINED. The MPI implementation will then raise an error of class
18 19	MPI_ERR_VALUE_TOO_LARGE.
20	
21	Datarep Conversion Functions
22	<pre>typedef int MPI_Datarep_conversion_function(void *userbuf,</pre>
23	MPI_Datatype datatype, int count, void *filebuf,
24	<pre>MPI_Offset position, void *extra_state);</pre>
25 26	<pre>typedef int MPI_Datarep_conversion_function_c(void *userbuf,</pre>
26	<pre>typedef int MPI_Datarep_conversion_function_c(void *userbuf, MPI_Datatype datatype, MPI_Count count, void *filebuf,</pre>
26 27	MPI_Datatype datatype, MPI_Count count, void *filebuf,
26 27 28	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state);</pre>
26 27 28 29	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror)</pre>
26 27 28 29 30 31 32	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR</pre>
26 27 28 29 30 31 32 33	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf</pre>
26 27 28 29 30 31 32 33 34	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype</pre>
26 27 28 29 30 31 32 33	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror</pre>
26 27 28 29 30 31 32 33 34 35	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER(KIND=MPI_OFFSET_KIND) :: position</pre>
26 27 28 29 30 31 32 33 34 35 36	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER (KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state</pre>
26 27 28 29 30 31 32 33 34 35 36 37	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER (KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state ABSTRACT INTERFACE</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER (KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count,</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER (KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count, filebuf, position, extra_state, ierror) !(_c)</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER(KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_OFFSET_KIND) :: extra_state ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count, filebuf, position, extra_state, ierror) !(_c) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER (KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count, filebuf, position, extra_state, ierror) !(_c)</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER(KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count, filebuf, position, extra_state, ierror) !(_c) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER (KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count, filebuf, position, extra_state, ierror) !(_c) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER(KIND=MPI_COUNT_KIND) :: count INTEGER(KIND=MPI_COFFSET_KIND) :: position</pre>
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45	<pre>MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state); ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count, filebuf, position, extra_state, ierror) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER :: count, ierror INTEGER (KIND=MPI_OFFSET_KIND) :: position INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state ABSTRACT INTERFACE SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count, filebuf, position, extra_state, ierror) !(_c) USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), VALUE :: userbuf, filebuf TYPE(MPI_Datatype) :: datatype INTEGER(KIND=MPI_COUNT_KIND) :: count</pre>

The function read\_conversion\_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in extra\_state, the argument that was passed to the MPI\_REGISTER\_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

Advice to users. Although the conversion functions have similarities to MPI\_PACK and MPI\_UNPACK, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI\_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

Advice to implementors. A converted read operation could be implemented as follows:

- 1. Get file extent of all data items
- 2. Allocate a filebuf large enough to hold all count data items
- 3. Read data from file into filebuf
- 4. Call read\_conversion\_fn to convert data and place it into userbuf
- 5. Deallocate filebuf

### (End of advice to implementors.)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

Rationale. Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion 48

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16

17

18 19

20

21

22

23

 $^{24}$ 

25 26

27 28

29

30

31 32

33

34 35

36 37

38

39

40

41

42

43

44

function can use the **position** to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

Advice to users. Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

9 The function write\_conversion\_fn must convert from native representation to file data 10 representation. Before calling this routine, MPI allocates filebuf of a size large enough to 11 hold **count** contiguous data items. The type of each data item matches the corresponding 12entry for the predefined datatype in the type signature of datatype. The function must copy 13 count data items from userbuf in the distribution described by datatype, to a contiguous 14distribution in filebuf, converting each data item from native representation to file repre-15sentation. If the size of datatype is less than the size of count data items, the conversion 16function must treat datatype as being contiguously tiled over the userbuf.

The function must distribut datatype as being contiguously there over the distribut. The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra\_state, the argument that was passed to the MPI\_REGISTER\_DATAREP call.

The predefined constant MPI\_CONVERSION\_FN\_NULL may be used as either write\_conversion\_fn or read\_conversion\_fn in bindings of MPI\_REGISTER\_DATAREP without large counts in these conversion callbacks, whereas the constant

<sup>24</sup> MPI\_CONVERSION\_FN\_NULL\_C can be used in the large count version (i.e.,

<sup>25</sup> MPI\_Register\_datarep\_c). In either of these cases, MPI will not attempt to invoke

write\_conversion\_fn or read\_conversion\_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section

(read\_conversion\_fn, write\_conversion\_fn, and dtype\_file\_extent\_fn) when one of the read or write routines in Section 14.4, or MPI\_FILE\_GET\_TYPE\_EXTENT is called by the user. dtype\_file\_extent\_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI\_SUCCESS, the implementation will raise an error in the class MPI\_ERR\_CONVERSION.

45

46

# 14.5.4 Matching Data Representations

<sup>47</sup> It is the user's responsibility to ensure that the data representation used to read data from <sup>48</sup> a file is *compatible* with the data representation that was used to write that data to the file.

1

2

3

4

5

6

7

In general, using the same data representation name when writing and reading a file does not guarantee that the representation is compatible. Similarly, using different representation names on two different implementations may yield compatible representations.

Compatibility can be obtained when "external32" representation is used, although precision may be lost and the performance may be less than when "native" representation is used. Compatibility is guaranteed using "external32" provided at least one of the following conditions is met.

- The data access routines directly use types enumerated in Section 14.5.2, that are supported by all implementations participating in the I/O. The predefined type used to write a data item must also be used to read a data item.
- In the case of Fortran 90 programs, the programs participating in the data accesses obtain compatible datatypes using MPI routines that specify precision and/or range (Section 19.1.9).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatibility with another implementation's "native" or "internal" representation.

Advice to users. Section 19.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

### Consistency and Semantics 14.6

#### 14.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file 30 accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses 34other than the above. Sequential consistency means the behavior of a set of operations will 35be as if the operations were performed in some serial order consistent with program order; each access appears atomic, although the exact ordering of accesses is unspecified. Userimposed consistency may be obtained using program order and calls to MPI\_FILE\_SYNC.

Let  $FH_1$  be the set of file handles created from one particular collective open of the 39 file FOO, and  $FH_2$  be the set of file handles created from a different collective open of 40 FOO. Note that nothing restrictive is said about  $FH_1$  and  $FH_2$ : the sizes of  $FH_1$  and 41  $FH_2$  may be different, the groups of processes used for each open may or may not intersect, 42the file handles in  $FH_1$  may be destroyed before those in  $FH_2$  are created, etc. Consider 43the following three cases: a single file handle (e.g.,  $fh_1 \in FH_1$ ), two file handles created 44from a single collective open (e.g.,  $fh_{1a} \in FH_1$  and  $fh_{1b} \in FH_1$ ), and two file handles from 45different collective opens (e.g.,  $fh_1 \in FH_1$  and  $fh_2 \in FH_2$ ). 46

For the purpose of consistency semantics, a matched pair (Section 14.4.5) of split col-47lective data access operations (e.g., MPI\_FILE\_READ\_ALL\_BEGIN and 48

1

2

3

4

 $\mathbf{5}$ 

6  $\overline{7}$ 

8

9

10

11 12

13

14

1516

17

18 19

20

21

22

23

 $^{24}$ 2526

2728

29

31

32

33

36

37

MPI\_FILE\_READ\_ALL\_END) compose a single data access operation. Similarly, a non blocking data access routine (e.g., MPI\_FILE\_IREAD) and the routine which completes the
 request (e.g., MPI\_WAIT) also compose a single data access operation. For all cases below,
 these data access operations are subject to the same constraints as blocking data access
 operations.

Advice to users. For an MPI\_FILE\_IREAD and MPI\_WAIT pair, the operation begins when MPI\_FILE\_IREAD is called and ends when MPI\_WAIT returns. (*End of advice to users.*)

Assume that  $A_1$  and  $A_2$  are two data access operations. Let  $D_1$  ( $D_2$ ) be the set of absolute byte displacements of every byte accessed in  $A_1$  ( $A_2$ ). The two data accesses *overlap* if  $D_1 \cap D_2 \neq \emptyset$ . The two data accesses *conflict* if they overlap and at least one is a write access.

Let  $SEQ_{fh}$  be a sequence of file operations on a single file handle, bracketed by

<sup>16</sup> MPI\_FILE\_SYNCs on that file handle. (Both opening and closing a file implicitly perform <sup>17</sup> an MPI\_FILE\_SYNC.)  $SEQ_{fh}$  is a "write sequence" if any of the data access operations in <sup>18</sup> the sequence are writes or if any of the file manipulation operations in the sequence change <sup>19</sup> the state of the file (e.g., MPI\_FILE\_SET\_SIZE or MPI\_FILE\_PREALLOCATE). Given two <sup>20</sup> sequences,  $SEQ_1$  and  $SEQ_2$ , we say they are not *concurrent* if one sequence is guaranteed <sup>21</sup> to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

25

<sup>26</sup> Case 1:  $fh_1 \in FH_1$  All operations on  $fh_1$  are sequentially consistent if atomic mode is <sup>27</sup> set. If nonatomic mode is set, then all operations on  $fh_1$  are sequentially consistent if they <sup>28</sup> are either nonconcurrent, nonconflicting, or both.

29

<sup>30</sup> Case 2:  $fh_{1a} \in FH_1$  and  $fh_{1b} \in FH_1$  Assume  $A_1$  is a data access operation using  $fh_{1a}$ , <sup>31</sup> and  $A_2$  is a data access operation using  $fh_{1b}$ . If for any access  $A_1$ , there is no access  $A_2$ <sup>32</sup> that conflicts with  $A_1$ , then MPI guarantees sequential consistency.

<sup>33</sup> However, unlike POSIX semantics, the default MPI semantics for conflicting accesses <sup>34</sup> do not guarantee sequential consistency. If  $A_1$  and  $A_2$  conflict, sequential consistency can <sup>35</sup> be guaranteed by either enabling atomic mode via the MPI\_FILE\_SET\_ATOMICITY routine, <sup>36</sup> or meeting the condition described in Case 3 below.

37 38

39

40

Case 3:  $fh_1 \in FH_1$  and  $fh_2 \in FH_2$  Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, MPI\_FILE\_SYNC must be used (both opening and closing a file implicitly perform an MPI\_FILE\_SYNC).

<sup>41</sup> Sequential consistency is guaranteed among accesses to a single file if for any write <sup>42</sup> sequence  $SEQ_1$  to the file, there is no sequence  $SEQ_2$  to the file which is *concurrent* with <sup>43</sup>  $SEQ_1$ . To guarantee sequential consistency when there are write sequences,

<sup>44</sup> MPI\_FILE\_SYNC must be used together with a mechanism that guarantees nonconcurrency
 <sup>45</sup> of the sequences.

46 See the examples in Section 14.6.11 for further clarification of some of these consistency
 47 semantics.

6 7

8 9

10

MPI_FILE_	SET_ATOMICITY(fh, flag)		1
INOUT	fh	file handle (handle)	2 3
IN	flag	true to set atomic mode, false to set nonatomic mode	4
	5	(logical)	5
			6
C binding	S		7
-	ile_set_atomicity(MPI_Fil	le fh, int flag)	8
Fortron 2	008 binding		9
	<pre>set_atomicity(fh, flag, f</pre>	ierror)	10
TYPE(MPI_File), INTENT(IN) :: fh			11
LOGICAL, INTENT(IN) :: flag			12 13
INTEG	INTEGER, OPTIONAL, INTENT(OUT) :: ierror		
Fortran b	inding		14 15
	SET_ATOMICITY(FH, FLAG, 1	(FRROR)	16
	ER FH, IERROR		17
	AL FLAG		18
Let E	II he the set of fle handles	masted by one collective open. The consistency	19
		created by one collective open. The consistency sing $FH$ is set by collectively calling	20
	-	MPI_FILE_SET_ATOMICITY is collective; all pro-	21
		alues for fh and flag. If flag is true, atomic mode is	22
	s false, nonatomic mode is se		23 24
Chang	ing the consistency semantic	s for an open file only affects new data accesses.	24
-	All completed data accesses are guaranteed to abide by the consistency semantics in effect		
during their execution. Nonblocking data accesses and split collective operations that have			27
not completed (e.g., via MPI_WAIT) are only guaranteed to abide by nonatomic mode			28
consistency semantics.			29
Advia	e to implementors Since the	semantics guaranteed by atomic mode are stronger	30
than those guaranteed by nonatomic mode, an implementation is free to adhere to			31
	the more stringent atomic mode semantics for outstanding requests. (End of advice		
	plementors.)		33 34
			35
			36
MPI_FILE_	GET_ATOMICITY(fh, flag)		37
IN –	fh	file handle (handle)	38
			39
OUT	flag	true if atomic mode, false if nonatomic mode (logical)	40
			41
C binding			42
int MPI_F	ile_get_atomicity(MPI_Fil	le in, int *ilag)	$43 \\ 44$
	008 binding		44 45
	get_atomicity(fh, flag, i		46
	MPI_File), INTENT(IN) ::	fh	47
LUGIC	AL, INTENT(OUT) :: flag		48

1	INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2 3 4 5 6	Fortran binding MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR LOGICAL FLAG
7 8 9 10	MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic mode is enabled; if flag is false, nonatomic mode is enabled.
11 12	MPI_FILE_SYNC(fh)
13 14	INOUT fh file handle (handle)
15 16 17	C binding int MPI_File_sync(MPI_File fh)
18 19 20 21	<pre>Fortran 2008 binding MPI_File_sync(fh, ierror)     TYPE(MPI_File), INTENT(IN) :: fh     INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>
22 23 24 25	Fortran binding MPI_FILE_SYNC(FH, IERROR) INTEGER FH, IERROR
26 27 28 29 30	Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of fh by the calling process. MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see above).
31 32 33 34 35	MPI_FILE_SYNC is a collective operation. The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SYNC—otherwise, the call to MPI_FILE_SYNC is erroneous.
36	14.6.2 Random Access vs. Sequential Files
<ol> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> </ol>	MPI distinguishes ordinary random access files from sequential stream files, such as pipes and tape files. Sequential stream files must be opened with the MPI_MODE_SEQUENTIAL flag set in the amode. For these files, the only permitted data access operations are shared file pointer reads and writes. Filetypes and etypes with holes are erroneous. In addition, the notion of file pointer is not meaningful; therefore, calls to MPI_FILE_SEEK_SHARED and MPI_FILE_GET_POSITION_SHARED are erroneous, and the pointer update rules specified for the data access routines do not apply. The amount of data accessed by a data access operation will be the amount requested unless the end of file is reached or an error is raised.
46 47 48	<i>Rationale.</i> This implies that reading on a pipe will always wait until the requested amount of data is available or until the process writing to the pipe has issued an end of file. ( <i>End of rationale.</i> )

Finally, for some sequential files, such as those corresponding to magnetic tapes or streaming network connections, writes to the file may be destructive. In other words, a write may act as a truncate (a MPI\_FILE\_SET\_SIZE with size set to the current position) followed by the write.

### 14.6.3 Progress

The *progress* rules of MPI are both a promise to users and a set of constraints on implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point-to-point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

## 14.6.4 Collective File Operations

Collective file operations are subject to the same restrictions as collective communication operations. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Collective file operations are collective over a duplicate of the communicator used to open the file—this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

### 14.6.5 Nonblocking Collective File Operations

Nonblocking collective file operations are defined only for data access routines with explicit offsets and individual file pointers but not with shared file pointers.

Nonblocking collective file operations are subject to the same restrictions as blocking collective I/O operations. All processes belonging to the group of the communicator that was used to open the file must call collective I/O operations (blocking and nonblocking) in the same order. This is consistent with the ordering rules for collective operations in threaded environments. For a complete discussion, please refer to the semantics set forth in Section 6.14.

Nonblocking collective I/O operations do not match with blocking collective I/O operations. Multiple nonblocking collective I/O operations can be outstanding on a single file handle. High quality MPI implementations should be able to support a large number of pending nonblocking I/O operations.

All nonblocking collective I/O calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation which may progress independently of any communication, computation, or I/O. The call returns a request handle, which must be passed to a completion call. Input buffers should not be modified and output buffers should not be accessed before the completion call returns. The same *progress* rules described 48

4 5 6

7

8

9

10

11

12

13

14

15

16

17

18

19

20 21

22 23

 $^{24}$ 

25

26

27

28

29 30

31 32

33

34

35

36 37

> 38 39

> 40 41

42

43

1

 $\mathbf{2}$ 

1 for nonblocking collective operations apply for nonblocking collective I/O operations. For  $\mathbf{2}$ a complete discussion, please refer to the semantics set forth in Section 6.12.

3 4

5

7

8

9

# 14.6.6 Type Matching

The type matching rules for I/O mimic the type matching rules for communication with one 6 exception: if etype is MPI\_BYTE, then this matches any datatype in a data access operation. In general, the etype of data items written must match the etype used to read the items, and for each data access operation, the current etype must also match the type declaration of the data access buffer. 10

11

1213

14

1516

17

Advice to users. In most cases, use of MPI\_BYTE as a wild card will defeat the file interoperability features of MPI. File interoperability can only perform automatic conversion between heterogeneous data representations when the exact datatypes accessed are explicitly specified. (End of advice to users.)

14.6.7 Miscellaneous Clarifications

18 Once an I/O routine completes, it is safe to free any opaque objects passed as arguments 19to that routine. For example, the comm and info used in an MPI\_FILE\_OPEN, or the etype 20and filetype used in an MPI\_FILE\_SET\_VIEW, can be freed without affecting access to the 21file. Note that for nonblocking routines and split collective operations, the operation must 22 be completed before it is safe to reuse data buffers passed as arguments.

23As in communication, datatypes must be committed before they can be used in file  $^{24}$ manipulation or data access operations. For example, the etype and filetype must be com-25mitted before calling MPI\_FILE\_SET\_VIEW, and the datatype must be committed before 26 calling MPI\_FILE\_READ or MPI\_FILE\_WRITE.

2728

29

33

### 14.6.8 MPI\_Offset Type

MPI\_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest 30  $^{31}$ file supported by MPI. Displacements and offsets are always specified as values of type MPI\_Offset. 32

In Fortran, the corresponding integer is an integer with kind parameter

34MPI\_OFFSET\_KIND, which is defined in the mpi\_f08 module, the mpi module and the mpif.h include file. 35

In Fortran 77 environments that do not support KIND parameters, MPI\_Offset arguments 36 37 should be declared as an INTEGER of suitable size. The language interoperability implications for MPI\_Offset are similar to those for addresses (see Section 19.3). 38

39 40

41

### 14.6.9 Logical vs. Physical File Layout

MPI specifies how the data should be laid out in a virtual file structure (the view), not 42how that file structure is to be stored on one or more disks. Specification of the physical 43 file structure was avoided because it is expected that the mapping of files to disks will be 44 system specific, and any specific control over file layout would therefore restrict program 45 portability. However, there are still cases where some information may be necessary to 46 optimize file layout. This information can be provided as *hints* specified via info when a file 47is created (see Section 14.2.8). 48

## 14.6.10 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI\_FILE\_SET\_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI\_FILE\_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI\_FILE\_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or MPI\_FILE\_OPEN, returned.

When applying consistency semantics, calls to MPI\_FILE\_SET\_SIZE and MPI\_FILE\_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and MPI\_FILE\_GET\_SIZE is considered a read of the file (which overlaps with all accesses to the file).

Advice to users. Any sequence of operations containing the collective routines MPI\_FILE\_SET\_SIZE and MPI\_FILE\_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 14.6.1 are satisfied. (*End of advice to users.*)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

Advice to users. Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI\_FILE\_READ of 10 bytes and an MPI\_FILE\_SET\_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (*End of advice to users.*)

## 14.6.11 Examples

The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address

- conflicting accesses on file handles obtained from a single collective open, and
- all accesses on file handles obtained from two separate collective opens.

The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of b will be 5. If nonatomic mode is set, the results of the read are undefined.

 $^{31}$ 

```
1
     /* Process 0 */
\mathbf{2}
3
     int i, a[10];
4
     int TRUE = 1;
\mathbf{5}
6
     for (i=0;i<10;i++)</pre>
7
        a[i] = 5;
8
9
     MPI_File_open(MPI_COMM_WORLD, "workfile",
10
                    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
11
     MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
12
     MPI_File_set_atomicity(fh0, TRUE);
13
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
14
     /* MPI_Barrier(MPI_COMM_WORLD); */
15
16
     /* Process 1 */
17
     int b[10];
18
19
     int TRUE = 1;
     MPI_File_open(MPI_COMM_WORLD, "workfile",
20
21
                    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
     MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
22
     MPI_File_set_atomicity(fh1, TRUE);
23
^{24}
     /* MPI_Barrier(MPI_COMM_WORLD); */
     MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
25
26
     A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
27
     temporal order with, for example, calls to MPI_BARRIER.
28
29
          Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
30
          order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
31
          received by process 1 using MPI_RECV. (End of advice to users.)
32
33
         Alternatively, a user can impose consistency with nonatomic mode set:
34
35
     /* Process 0 */
36
     int i, a[10];
37
     for (i=0;i<10;i++)</pre>
38
        a[i] = 5;
39
40
     MPI_File_open(MPI_COMM_WORLD, "workfile",
41
                    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
42
     MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
43
     MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
44
     MPI_File_sync(fh0);
45
     MPI_Barrier(MPI_COMM_WORLD);
46
     MPI_File_sync(fh0);
47
48
     /* Process 1 */
```

```
1
                                                                                      \mathbf{2}
int b[10];
                                                                                      3
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                      4
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                      5
                                                                                       6
MPI_File_sync(fh1);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_sync(fh1);
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
                                                                                      9
                                                                                      10
The "sync-barrier-sync" construct is required because:
                                                                                      11
                                                                                      12
   • The barrier ensures that the write on process 0 occurs before the read on process 1.
                                                                                      13
   • The first sync guarantees that the data written by all processes is transferred to the
                                                                                      14
     storage device.
                                                                                      15
                                                                                      16
   • The second sync guarantees that all data which has been transferred to the storage
                                                                                      17
     device is visible to all processes. (This does not affect process 0 in this example.)
                                                                                      18
                                                                                      19
    The following program represents an erroneous attempt to achieve consistency by elim-
                                                                                      20
inating the apparently superfluous second "sync" call for each process.
                                                                                      21
/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
                                                                                      22
/* Process 0 */
                                                                                      23
                                                                                      ^{24}
int i, a[10];
                                                                                      25
for (i=0;i<10;i++)
                                                                                      26
   a[i] = 5;
                                                                                      27
                                                                                      28
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                      29
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
                                                                                      30
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                      31
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
                                                                                      32
MPI_File_sync(fh0);
                                                                                      33
MPI_Barrier(MPI_COMM_WORLD);
                                                                                      34
                                                                                      35
/* Process 1 */
                                                                                      36
                                                                                      37
int b[10];
                                                                                      38
MPI_File_open(MPI_COMM_WORLD, "workfile",
                                                                                      39
               MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
                                                                                      40
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
                                                                                      41
MPI_Barrier(MPI_COMM_WORLD);
                                                                                      42
MPI_File_sync(fh1);
                                                                                      43
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
                                                                                      44
                                                                                      45
/* ----- THIS EXAMPLE IS ERRONEOUS ----- */
                                                                                      46
                                                                                      47
```

The above program also violates the MPI rule against out-of-order collective operations and will deadlock for implementations in which MPI\_FILE\_SYNC blocks.

1 Advice to users. Some implementations may choose to implement MPI\_FILE\_SYNC  $\mathbf{2}$ as a temporally synchronizing function. When using such an implementation, the 3 "sync-barrier-sync" construct above can be replaced by a single "sync." The results of 4 using such code with an implementation for which MPI\_FILE\_SYNC is not temporally  $\mathbf{5}$ synchronizing is undefined. (*End of advice to users.*) 6 7Asynchronous I/O 8 The behavior of asynchronous I/O operations is determined by applying the rules specified 9 above for synchronous I/O operations. 10 The following examples all access a preexisting file "myfile." Word 10 in myfile initially 11 contains the integer 2. Each example writes and reads word 10. 12First consider the following code fragment: 13 14int a = 4, b, TRUE=1; 15MPI\_File\_open(MPI\_COMM\_WORLD, "myfile", 16MPI\_MODE\_RDWR, MPI\_INFO\_NULL, &fh); 17MPI\_File\_set\_view(fh, 0, MPI\_INT, MPI\_INT, "native", MPI\_INFO\_NULL); 18/\* MPI\_File\_set\_atomicity(fh, TRUE); Use this to set atomic mode. \*/ 19MPI\_File\_iwrite\_at(fh, 10, &a, 1, MPI\_INT, &reqs[0]); 20MPI\_File\_iread\_at(fh, 10, &b, 1, MPI\_INT, &reqs[1]); 21MPI\_Waitall(2, regs, statuses); 2223For asynchronous data access operations, MPI specifies that the access occurs at any time  $^{24}$ between the call to the asynchronous data access routine and the return from the corre-25sponding request complete routine. Thus, executing either the read before the write, or the 26write before the read is consistent with program order. If atomic mode is set, then MPI 27guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic 28mode is not set, then sequential consistency is not guaranteed and the program may read  $^{29}$ something other than 2 or 4 due to the conflicting data access. 30 Similarly, the following code fragment does not order file accesses:  $^{31}$ int a = 4, b; 32MPI\_File\_open(MPI\_COMM\_WORLD, "myfile", 33 MPI\_MODE\_RDWR, MPI\_INFO\_NULL, &fh); 34MPI\_File\_set\_view(fh, 0, MPI\_INT, MPI\_INT, "native", MPI\_INFO\_NULL); 35 /\* MPI\_File\_set\_atomicity(fh, TRUE); Use this to set atomic mode. \*/ 36 MPI\_File\_iwrite\_at(fh, 10, &a, 1, MPI\_INT, &reqs[0]); 37 MPI\_File\_iread\_at(fh, 10, &b, 1, MPI\_INT, &reqs[1]); 38 MPI\_Wait(&reqs[0], &status); 39 MPI\_Wait(&reqs[1], &status); 40 $^{41}$ If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee 42sequential consistency in nonatomic mode. 43On the other hand, the following code fragment: 4445int a = 4, b; 46MPI\_File\_open(MPI\_COMM\_WORLD, "myfile", 47MPI\_MODE\_RDWR, MPI\_INFO\_NULL, &fh); 48MPI\_File\_set\_view(fh, 0, MPI\_INT, MPI\_INT, "native", MPI\_INFO\_NULL);

<pre>MPI_File_iwrite_at(fh, 10, &amp;a, 1, MPI_INT, &amp;reqs[0]);</pre>	1
<pre>MPI_Wait(&amp;reqs[0], &amp;status);</pre>	2
<pre>MPI_File_iread_at(fh, 10, &amp;b, 1, MPI_INT, &amp;reqs[1]);</pre>	3
<pre>MPI_Wait(&amp;reqs[1], &amp;status);</pre>	4 5
defines the same ordering as:	6
int a = 4, b;	7
MPI_File_open(MPI_COMM_WORLD, "myfile",	8
MPI_MODE_RDWR, MPI_INFO_NULL, &fh);	9
<pre>MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);</pre>	10
<pre>MPI_File_write_at(fh, 10, &amp;a, 1, MPI_INT, &amp;status );</pre>	11
<pre>MPI_File_read_at(fh, 10, &amp;b, 1, MPI_INT, &amp;status );</pre>	12 13
Since	13
- nonconcurrent enceptions on a single fle handle are securetially consistent and	15
• nonconcurrent operations on a single file handle are sequentially consistent, and	16
• the program fragments specify an order for the operations,	17
MPI guarantees that both program fragments will read the value 4 into b. There is no need	18
to set atomic mode for this example.	19
Similar considerations apply to conflicting accesses of the form:	20
	21 22
<pre>MPI_File_iwrite_all(fh,); MPI_File_iread_all(fh,);</pre>	22
MPI_Waitall();	24
	25
In addition, as mentioned in Section 14.6.5, nonblocking collective I/O operations have	26
to be called in the same order on the file handle by all processes.	27
Similar considerations apply to conflicting accesses of the form:	28
<pre>MPI_File_write_all_begin(fh,);</pre>	29
<pre>MPI_File_iread(fh,);</pre>	30
MPI_Wait(fh,);	31 32
<pre>MPI_File_write_all_end(fh,);</pre>	33
Recall that constraints governing consistency and semantics are not relevant to the	34
following:	35
<pre>MPI_File_write_all_begin(fh,);</pre>	36
<pre>MPI_File_read_all_begin(fh,);</pre>	37
<pre>MPI_File_read_all_end(fh,);</pre>	38
<pre>MPI_File_write_all_end(fh,);</pre>	39
since split collective operations on the same file handle may not overlap (see Section 14.4.5).	40 41
	41
14.7 I/O Error Handling	43
	44
By default, communication errors are fatal—MPI_ERRORS_ARE_FATAL is the default error	45
handler associated with $MPI_COMM_WORLD$ . I/O errors are usually less catastrophic (e.g.,	46

By default, communication errors are fatal—MPI\_ERRORS\_ARE\_FATAL is the default error <sup>45</sup> handler associated with MPI\_COMM\_WORLD. I/O errors are usually less catastrophic (e.g., <sup>46</sup> "file not found") than communication errors, and common practice is to catch these errors <sup>47</sup> and continue executing. For this reason, MPI provides additional error facilities for I/O. <sup>48</sup> Advice to users. MPI does not specify the state of a computation after an erroneous MPI call has occurred. A high-quality implementation will support the I/O error handling facilities, allowing users to write programs using common practice for I/O. (End of advice to users.)

Like communicators, each file handle has an error handler associated with it. The MPI I/O error handling routines are defined in Section 9.3.

<sup>8</sup> When MPI calls a user-defined error handler resulting from an error on a particular <sup>9</sup> file handle, the first two arguments passed to the file error handler are the file handle and <sup>10</sup> the error code. For I/O errors that are not associated with a valid file handle (e.g., in <sup>11</sup> MPI\_FILE\_OPEN or MPI\_FILE\_DELETE), the first argument passed to the error handler is <sup>12</sup> MPI\_FILE\_NULL.

I/O error handling differs from communication error handling in another important 13 aspect. By default, the predefined error handler for file handles is MPI\_ERRORS\_RETURN. 14The **default file error** handler has two purposes: when a new file handle is created (by 1516MPI\_FILE\_OPEN), the error handler for the new file handle is initially set to the default 17file error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI\_FILE\_OPEN or MPI\_FILE\_DELETE) use the default file error handler. The 18 19default file error handler can be changed by specifying MPI\_FILE\_NULL as the fh argument to MPI\_FILE\_SET\_ERRHANDLER. The current value of the default file error handler can 20be determined by passing MPI\_FILE\_NULL as the fh argument to 21MPI\_FILE\_GET\_ERRHANDLER.

22 23 24

25

26

27

Rationale. For communication, the default error handler is inherited from

MPI\_COMM\_WORLD when using the World Model. In I/O, there is no analogous "root" file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI\_FILE\_NULL. (*End of rationale.*)

28 29 30

31 32

33

34

35

36 37

38 39

40 41

43

# 14.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 14.5.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI\_ERR\_TYPE.

# 14.9 Examples

```
14.9.1 Double Buffering with Split Collective I/O
```

This example shows how to overlap computation and output. The computation is performed
 by the function compute\_buffer().

1

2

3

4

5 6

 $\overline{7}$ 

MPI_ERR_FILE	Invalid file handle	1
MPI_ERR_NOT_SAME	Collective argument not identical on all	2
	processes, or collective routines called in	3
	a different order by different processes	4
MPI_ERR_AMODE	Error related to the <b>amode</b> passed to	5
	MPI_FILE_OPEN	6
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to	7
	MPI_FILE_SET_VIEW	8
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on	9
	a file which supports sequential access only	10
MPI_ERR_NO_SUCH_FILE	File does not exist	11
MPI_ERR_FILE_EXISTS	File exists	12
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)	13
MPI_ERR_ACCESS	Permission denied	14
MPI_ERR_NO_SPACE	Not enough space	15
MPI_ERR_QUOTA	Quota exceeded	16
MPI_ERR_READ_ONLY	Read-only file or file system	17
MPI_ERR_FILE_IN_USE	File operation could not be completed, as	18
	the file is currently open by some process	19
MPI_ERR_DUP_DATAREP	Conversion functions could not be regis-	20
	tered because a data representation identi-	21
	fier that was already defined was passed to	22
	MPI_REGISTER_DATAREP	23
MPI_ERR_CONVERSION	An error occurred in a user supplied data	24
	conversion function.	25
MPI_ERR_IO	Other I/O error	26
		27
Table 14.5	5: I/O Error Classes	28

```
1
     *
           void double_buffer(
\mathbf{2}
                                                            ** IN
      *
                    MPI_File fh,
3
      *
                    MPI_Datatype buftype,
                                                            ** IN
4
      *
                    int bufcount
                                                             ** IN
5
      *
            )
6
     *
7
     * Description:
8
            Performs the steps to overlap computation with a collective write
      *
9
            by using a double-buffering technique.
      *
10
      *
11
      * Parameters:
12
      *
           fh
                            previously opened MPI file han
MPI datatype for memory layout
                               previously opened MPI file handle
13
            buftype
     *
14
      *
                              (Assumes a compatible view has been set on fh)
15
                              # buftype elements to transfer
      *
            bufcount
16
      *-----*/
17
18
     /* this macro switches which buffer "x" is pointing to */
19
    #define TOGGLE_PTR(x) (((x)==(buffer1)) ? (x=buffer2) : (x=buffer1))
20
21
    void double_buffer(MPI_File fh, MPI_Datatype buftype, int bufcount)
22
     ſ
23
^{24}
       MPI_Status status; /* status for MPI calls */
       float *buffer1, *buffer2; /* buffers to hold results */
25
26
       float *compute_buf_ptr; /* destination buffer */
27
                                  /* for computing */
       float *write_buf_ptr; /* source for writing */
28
29
                                 /* determines when to quit */
       int done;
30
^{31}
       /* buffer initialization */
32
       buffer1 = (float *)
33
                          malloc(bufcount*sizeof(float));
34
       buffer2 = (float *)
35
                          malloc(bufcount*sizeof(float));
36
        compute_buf_ptr = buffer1; /* initially point to buffer1 */
37
       write_buf_ptr = buffer1; /* initially point to buffer1 */
38
39
40
        /* DOUBLE-BUFFER prolog:
41
            compute buffer1; then initiate writing buffer1 to disk
        *
42
        */
        compute_buffer(compute_buf_ptr, bufcount, &done);
43
44
       MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
45
       /* DOUBLE-BUFFER steady state:
46
47
        * Overlap writing old results from buffer pointed to by write_buf_ptr
48
        * with computing new results into buffer pointed to by compute_buf_ptr.
```

```
*
    *
       There is always one write-buffer and one compute-buffer in use
       during steady state.
    *
    */
  while (!done) {
      TOGGLE_PTR(compute_buf_ptr);
      compute_buffer(compute_buf_ptr, bufcount, &done);
      MPI_File_write_all_end(fh, write_buf_ptr, &status);
      TOGGLE_PTR(write_buf_ptr);
      MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
  }
   /* DOUBLE-BUFFER epilog:
    *
        wait for final write to complete.
    */
  MPI_File_write_all_end(fh, write_buf_ptr, &status);
  /* buffer cleanup */
  free(buffer1);
  free(buffer2);
}
```



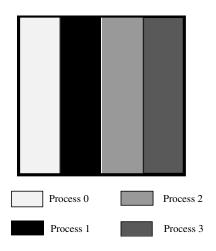


Figure 14.4: Example array file layout

Assume we are writing out a  $100 \times 100$  2D array of double precision floating point numbers that is distributed among 4 processes such that each process has a block of 25 columns (e.g., process 0 has columns 0–24, process 1 has columns 25–49, etc.; see Figure 14.4). To create the filetypes for each process one could use the following C program (see Section 5.1.3):

```
double subarray[100][25];
```

723

1

 $\mathbf{2}$ 

3

4

5 6

7

8

9 10

11 12

13 14

15

16

17 18

19 20

21

22 23 24

37

38 39

40 41

42

43

44

45

 $46 \\ 47$ 

```
1
\mathbf{2}
3
4
5
6
7
8
9
                                         MPI_DOUBLE
                                                           Holes
10
11
                       Figure 14.5: Example local array filetype for process 1
12
13
         MPI_Datatype filetype;
14
         int sizes[2], subsizes[2], starts[2];
15
         int rank;
16
17
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18
         sizes[0]=100; sizes[1]=100;
19
         subsizes[0]=100; subsizes[1]=25;
20
         starts[0]=0; starts[1]=rank*subsizes[1];
21
22
         MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
23
                                     MPI_DOUBLE, &filetype);
^{24}
25
          Or, equivalently in Fortran:
26
27
     double precision subarray(100,25)
28
     integer filetype, rank, ierror
29
     integer sizes(2), subsizes(2), starts(2)
30
^{31}
     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
32
     sizes(1)
                   = 100
33
     sizes(2)
                   = 100
34
     subsizes(1) = 100
35
     subsizes(2) = 25
36
     starts(1)
                   = 0
37
     starts(2)
                   = rank*subsizes(2)
38
39
     call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
40
                  MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,
                                                                        &
^{41}
                  filetype, ierror)
42
43
          The generated filetype will then describe the portion of the file contained within the
44
     process's subarray with holes for the space taken by the other processes. Figure 14.5 shows
45
     the filetype created for process 1.
46
```

- 47
- 48

# Chapter 15

# **Tool Support**

# 15.1 Introduction

This chapter discusses interfaces that allow debuggers, performance analyzers, and other tools to extract information about the operation of MPI processes. Specifically, this chapter defines both the MPI profiling interface (Section 15.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 15.3), which supports the inspection and manipulation of MPI control and performance variables, as well as the registration of callbacks for MPI library events. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

# 15.2 Profiling Interface

# 15.2.1 Requirements

To meet the requirements for the  $\mathsf{MPI}$  profiling interface, an implementation of the  $\mathsf{MPI}$  functions must

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI\_ for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the PMPI\_ version be supplied and work as expected, but it is not possible to replace at link time the MPI\_ version with a user-defined version.

For Fortran, the different support methods cause several specific procedure names. Therefore, several profiling routines (with these specific procedure names) are needed for each Fortran MPI routine, as described in Section 19.1.5.

- 2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
- document the implementation of different language bindings of the MPI interface if
   they are layered on top of each other, so that the profiler developer knows whether to
   implement the profile interface for each binding, or to economize by implementing it
   document the lowest level routines.

4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of "wrapper" functions that call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

This separability is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

 $11 \\ 12$ 

15

1

2

3

4

5

6

7

8

9

10

 $13 \\ 14$ 

15.2.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.

5. provide a no-op routine MPI\_PCONTROL in the MPI library.

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code that implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish *without* access to the underlying implementation.

We believe that having such an interface is important if MPI is to be attractive to end users, since the availability of many different tools will be a significant factor in attracting users to the MPI standard.

The profiling interface is just that, an interface. It says *nothing* about the way in which it is used. There is therefore no attempt to lay down what information is collected through the interface, or how the collected information is saved, filtered, or displayed.

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to it being used wherever it is useful.

As the issues being addressed here are intimately tied up with the way in which executable images are built, which may differ greatly on different machines, the examples given below should be treated solely as one way of implementing the objective of the MPI profiling interface. The actual requirements made of an implementation are those detailed in the Requirements section above, the whole of the rest of this section is only present as justification and discussion of the logic for those requirements.

The examples below show one way in which an implementation could be constructed to meet the requirements on a Unix system (there are doubtless others that would be equally valid).

44

<sup>45</sup> 15.2.3 Logic of the Design

<sup>46</sup>
 <sup>47</sup>
 <sup>47</sup>
 <sup>48</sup>
 <sup>48</sup>
 <sup>46</sup> Provided that an MPI implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept the MPI calls that are made by the

user program. The profiling system implementor can then collect any required information before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

## 15.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at noncritical points in the calculation.
- Adding user events to a trace file.

These requirements are met by use of MPI\_PCONTROL.

MPI\_PCONTROL(level, ...)

IN	level	Profiling level (integer)

### C binding

int MPI\_Pcontrol(const int level, ...)

Fortran 2008 binding MPI\_Pcontrol(level) INTEGER, INTENT(IN) :: level

### Fortran binding

MPI\_PCONTROL(LEVEL) INTEGER LEVEL

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics that will be provided by calls to MPI\_PCONTROL. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of level.

- level==0 Profiling is disabled.
- level==1 Profiling is enabled at a normal default level of detail.
- level==2 Profile buffers are flushed, which may be a no-op in some profilers.
- All other values of level have profile library defined effects and additional arguments.

We also request that the default state after MPI has been initialized is for profiling to <sup>45</sup> be enabled at the normal default level. (i.e., as if MPI\_PCONTROL had just been called <sup>46</sup> with the argument 1). This allows users to link with a profiling library and to obtain profile <sup>47</sup> output without having to modify their source code at all. <sup>48</sup>

1

 $\mathbf{2}$ 

3

4 5

6

7

8 9

10

11 12

13 14

15 16

17 18

19 20

21

22

23

24

25 26

27

28

29

30

31

32

33

34

35

36

37 38

39 40

41

42 43

The provision of MPI\_PCONTROL as a no-op in the standard MPI library supports the collection of more detailed profiling information with source code that can still link against the standard MPI library.

**Example 15.1** A wrapper to accumulate the total amount of data sent by the MPI\_SEND function, along with the total elapsed time spent in the function.

```
static int totalBytes = 0;
static double totalTime = 0.0;
int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
{
                                      /* Pass on all arguments */
   double tstart = MPI_Wtime();
   int size;
                 = PMPI_Send(buffer,count,datatype,dest,tag,comm);
   int result
   totalTime += MPI_Wtime() - tstart;
                                                /* and time
                                                                     */
   MPI_Type_size(datatype, &size); /* Compute size */
   totalBytes += count*size;
   return result;
}
```

#### MPI Library Implementation 15.2.5

If the MPI library is implemented in C on a Unix system, then there are various options, including the two presented here, for supporting the name-shift requirement. The choice between these two options depends partly on whether the linker and compiler support weak symbols.

If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other System V.4 machines), then only a single library is required as the following example shows:

**Example 15.2** Library implementation using weak symbols.

```
#pragma weak MPI_Example = PMPI_Example
int PMPI_Example(/* appropriate args */)
{
    /* Useful content */
}
```

The effect of this **#pragma** is to define the external symbol MPI\_Example as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library); however if no other definition exists, then the linker will use the weak definition.

46In the absence of weak symbols then one possible solution would be to use the C macro preprocessor as the following example shows:

1

 $\mathbf{2}$ 

3

4

5

6 7

8

9 10

11

12

13

14

15

1617

18 19

20

2122

23

 $^{24}$ 25

2627

28

29

30

 $^{31}$ 

32

33 34

35

36 37

38

39

40

41 42

43

44

45

47

```
Example 15.3 Library implementation using C pre-processor macros.
#ifdef PROFILELIB
#
     ifdef __STDC__
#
          define FUNCTION(name) P##name
#
     else
          define FUNCTION(name) P/**/name
#
#
     endif
#else
     define FUNCTION(name) name
#
#endif
Each of the user visible functions in the library would then be declared thus
```

```
int FUNCTION(MPI_Example)(/* appropriate args */)
{
    /* Useful content */
}
```

The same source file can then be compiled to produce both versions of the library, depending on the state of the **PROFILELIB** macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that need to be intercepted, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

# % cc ... -lmyprof -lpmpi -lmpi

Here libmyprof.a contains the profiler functions that intercept some of the MPI functions, libpmpi.a contains the "name shifted" MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

# 15.2.6 Complications

# Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI func-38 tions (e.g., a portable implementation of the collective operations implemented using point-39 to-point communications), there is potential for profiling functions to be called from within 40 an MPI function that was called from a profiling function. This could lead to "double 41 counting" of the time spent in the inner routine. Since this effect could actually be useful 42under some circumstances (e.g., it might allow one to answer the question "How much time 43 is spent in the point-to-point routines when they are called from collective functions?"), we 44have decided not to enforce any restrictions on the author of the MPI library that would 45overcome this. Therefore the author of the profiling library should be aware of this problem, 46and guard against it. In a single-threaded world this is easily achieved through use of a 47static variable in the profiling code that remembers if you are already inside a profiling 48

1

 $\mathbf{2}$ 

3

4

5

6

7

9

10

11 12

13 14

15

16

17

18 19

20

21

22

23

 $^{24}$ 

25

26

27 28

29 30

31

32

33 34

35 36

routine. It becomes more complex in a multithreaded environment (as does the meaning of
 the times recorded).

3

# <sup>4</sup> Linker Oddities

The Unix linker traditionally operates in one pass: the effect of this is that functions from libraries are only included in the image if they are needed at the time the library is scanned. When combined with weak symbols, or multiple definitions of the same function, this can cause odd (and unexpected) effects.

Consider, for instance, an implementation of MPI in which the Fortran binding is 10 achieved by using wrapper functions on top of the C implementation. The author of the 11 profile library then assumes that it is reasonable only to provide profile functions for the C 12binding, since Fortran will eventually call these, and the cost of the wrappers is assumed 13 to be small. However, if the wrapper functions are not in the profiling library, then none 14of the profiled entry points will be undefined when the profiling library is called. Therefore 15none of the profiling code will be included in the image. When the standard MPI library 16is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of 17the MPI functions. The overall effect is that the code will link successfully, but will not be 18 profiled. 19

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be copied out of the base library and into the profiling one using a tool such as **ar**.

24

# 25 Fortran Support Methods

The different Fortran support methods and possible options for the support of subarrays (depending on whether the compiler can support TYPE(\*), DIMENSION(...) choice buffers) imply different specific procedure names for the same Fortran MPI routine. The rules and implications for the profiling interface are described in Section 19.1.5.

30 31

32

# 15.2.7 Multiple Levels of Interception

The scheme given here does not directly support the nesting of profiling functions, since it provides only a single alternative name for each MPI function. Consideration was given to an implementation that would allow multiple levels of call interception, however we were unable to construct an implementation of this that did not have the following disadvantages

- assuming a particular implementation language, and
- 38 39 40

48

37

• imposing a run time cost even when no profiling was taking place.

Since one of the objectives of MPI is to permit efficient, low latency implementations, and
 it is not the business of a standard to require a particular implementation language, we
 decided to accept the scheme outlined above.

<sup>44</sup> Note, however, that it is possible to use the scheme above to implement a multi-level <sup>45</sup> system, since the function called by the user may call many different profiling functions <sup>46</sup> before calling the underlying MPI function. This capability has been demonstrated in the <sup>47</sup> P<sup>N</sup>MPI tool infrastructure [58].

# 15.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance and rely on internal events for their implementation. Understanding and manipulating these variables and tracking these events can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation, as well as expose events that can be tracked by tools. The interface is split into three parts: the first part provides information about, and supports the setting of, control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation. The third part enables tools to query available events within an MPI implementation and register callbacks for them.

To avoid restrictions on the MPI implementation, the MPI tool information interface 15allows the implementation to specify which control variables, performance variables, and 16events exist. Additionally, the user of the MPI tool information interface can obtain meta-17 data about each available variable or event, such as its datatype, and a textual description. 18 The MPI tool information interface provides the necessary routines to find all variables and 19events that exist in a particular MPI implementation; to query their properties; to retrieve 20descriptions about their meaning; to access and, if appropriate, to alter their values; and 21(in case of events) set callbacks triggered by them. 22

Variables, events, and categories across connected MPI processes with equivalent names 23are required to have the same meaning (see the definition of "equivalent" as related to strings  $^{24}$ in Section 15.3.3). Furthermore, enumerations with equivalent names across connected MPI 25processes are required to have the same meaning, but are allowed to comprise different 26enumeration items. Enumeration items that have equivalent names across connected MPI 27processes in enumerations with the same meaning must also have the same meaning. In  $^{28}$ order for variables and categories to have the same meaning, routines in the tools information 29interface that return details for those variables and categories have requirements on what 30 parameters must be identical. These requirements are specified in their respective sections. 31

*Rationale.* The intent of requiring the same meaning for entities with equivalent names is to enforce consistency across connected MPI processes. For example, variables describing the number of packets sent on different types of network devices should have different names to reflect their potentially different meanings. (*End of rationale.*)

The MPI tool information interface can be used independently from the MPI communication functionality. In particular, the routines of this interface can be called before MPI is initialized and after MPI is finalized. In order to support this behavior cleanly, the MPI tool information interface uses separate initialization and finalization routines. All identifiers used in the MPI tool information interface have the prefix MPI\_T\_.

On success, all MPI tool information interface routines return MPI\_SUCCESS, otherwise they return an appropriate and unique return code indicating the reason why the call was not successfully completed. Details on return codes can be found in Section 15.3.10. However, unsuccessful calls to the MPI tool information interface are not fatal and do not impact the execution of subsequent MPI routines.

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

11

12

13

14

32

33

34

35

36

37 38

39

40

41

42

43

44

45

46

47

<sup>1</sup> Since the MPI tool information interface primarily focuses on tools and support li-<sup>2</sup> braries, MPI implementations are only required to provide C bindings for functions and <sup>3</sup> constants introduced in this section. Except where otherwise noted, all conventions and <sup>4</sup> principles governing the C bindings of the MPI API also apply to the MPI tool information <sup>5</sup> interface, which is available by including the mpi.h header file. All routines in this interface <sup>6</sup> have local semantics.

Advice to users. The number and type of control variables, performance variables, and events can vary between MPI implementations, platforms and different builds of the same implementation on the same platform as well as between runs. Hence, any application relying on a particular variable will not be portable. Further, there is no guarantee that the number of variables and variable indices are the same across connected MPI processes.

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable or of a particular event. (*End of advice to users.*)

15.3.1 Verbosity Levels

The MPI tool information interface provides access to internal configuration and perfor-22 mance information through a set of control and performance variables defined by the MPI 23 $^{24}$ implementation. Since some implementations may export a large number of variables, 25variables are classified by a verbosity level that categorizes both their intended audience 26(end users, performance tuners or MPI implementors) and a relative measure of level of detail (basic, detailed or all). These verbosity levels are described by a single integer. 27Table 15.1 lists the constants for all possible verbosity levels. The values of the con-28stants are monotonic in the order listed in the table; i.e., MPI\_T\_VERBOSITY\_USER\_BASIC 29 < MPI\_T\_VERBOSITY\_USER\_DETAIL < ... < MPI\_T\_VERBOSITY\_MPIDEV\_ALL. 30

31		
32 MPI_T	_VERBOSITY_USER_BASIC	Basic information of interest to users
33 MPI_T	_VERBOSITY_USER_DETAIL	Detailed information of interest to users
34 MPI_T	_VERBOSITY_USER_ALL	All remaining information of interest to users
35 MPI_T	_VERBOSITY_TUNER_BASIC	Basic information required for tuning
36 MPI_T	_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
37 MPI_T	_VERBOSITY_TUNER_ALL	All remaining information required for tuning
38 MPI_T	_VERBOSITY_MPIDEV_BASIC	Basic information for MPI implementors
39 MPI_T	_VERBOSITY_MPIDEV_DETAIL	Detailed information for MPI implementors
40 MPI_T	_VERBOSITY_MPIDEV_ALL	All remaining information for MPI implementors

Table 15.1: MPI tool information interface verbosity levels

# 15.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

Each MPI tool information interface variable provides access to a particular control setting
 or performance property of the MPI implementation. A variable may refer to a specific

7

8

9

10 11

12

13 14

15

16

17

18

19 20

21

41 42

43 44 45

MPI object such as a communicator, datatype, or one-sided communication window, or the variable may refer more generally to the MPI environment of the process. Except for the last case, the variable must be bound to exactly one MPI object before it can be used. Table 15.2 lists all MPI object types to which an MPI tool information interface variable can be bound, together with the matching constant that MPI tool information interface routines return to identify the object type.

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMM	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OP	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WIN	MPI windows for one-sided communication
MPI_T_BIND_MPI_MESSAGE	MPI message object
MPI_T_BIND_MPI_INFO	MPI info object
MPI_T_BIND_MPI_SESSION	MPI session object

Table 15.2: Constants to identify associations of variables

*Rationale.* Some variables have meanings tied to a specific MPI object. Examples include the number of send or receive operations that use a particular datatype, the number of times a particular error handler has been called, or the communication protocol and "eager limit" used for a particular communicator. Creating a new MPI tool information interface variable for each MPI object would cause the number of variables to grow without bound, since they cannot be reused to avoid naming conflicts. By associating MPI tool information interface variables with a specific MPI object, the MPI implementation only must specify and maintain a single variable, which can then be applied to as many MPI objects of the respective type as created during the program's execution. (*End of rationale.*)

### 15.3.3 Convention for Returning Strings

Several MPI tool information interface functions return one or more strings. These functions have two arguments for each string to be returned: an OUT parameter that identifies a pointer to the buffer in which the string will be returned, and an INOUT parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function writes at most n-1 of the string's characters into the buffer, followed by a null terminator. If the returned string's length is greater than or equal to n, the string will be truncated to n-1 characters. In this case, the length of the string plus one (for the terminating null character) is returned in the length argument. If the user passes the null pointer as the buffer argument or passes 0 as the length argument, the function does not return the string and only returns the length of the 

 $\mathbf{2}$ 

 $\mathbf{5}$ 

string plus one in the length argument. If the user passes the null pointer as the length argument, the buffer argument is ignored and nothing is returned.

3 MPI implementations behave as if they have an internal character array that is copied 4 to the output character array supplied by the user. Such output strings are only defined 5to be equivalent if their notional source-internal character arrays are identical (up to and 6 including the null terminator), even if the output string is truncated due to a small input  $\overline{7}$ length parameter n.

8 9

10

1

 $\mathbf{2}$ 

# 15.3.4 Initialization and Finalization

The MPI tool information interface requires a separate set of initialization and finalization 11routines. 12

13 14

# MPI\_T\_INIT\_THREAD(required, provided)

15			
16	IN	required	desired level of thread support (integer)
17	OUT	provided	provided level of thread support (integer)

18 19

21

42

48

#### C binding 20

int MPI\_T\_init\_thread(int required, int \*provided)

All programs or tools that use the MPI tool information interface must initialize the 22MPI tool information interface in the processes that will use the interface before calling 23any other of its routines. A user can initialize the MPI tool information interface by calling  $^{24}$ MPI\_T\_INIT\_THREAD, which can be called multiple times. In addition, this routine initial-25izes the thread environment for all routines in the MPI tool information interface. Calling 26this routine when the MPI tool information interface is already initialized has no effect 27beyond increasing the reference count of how often the interface has been initialized. The 28argument required is used to specify the desired level of thread support. The possible values 29and their semantics are identical to the ones that can be used with MPI\_INIT\_THREAD 30 listed in Section 11.6. The call returns in provided information about the actual level of  $^{31}$ thread support that will be provided by the MPI implementation for calls to MPI tool 32 information interface routines. It can be one of the four values listed in Section 11.6. 33

The MPI specification does not require all MPI processes to exist before MPI is initial-34ized. If the MPI tool information interface is used before initialization of MPI, the user is 35 responsible for ensuring that the MPI tool information interface is initialized on all processes 36 it is used in. Processes created by the MPI implementation during initialization inherit the 37 status of the MPI tool information interface (whether it is initialized or not as well as all 38 active sessions and handles) from the process from which they are created. 39

Processes created at runtime as a result of calls to MPI's dynamic process management 40require their own initialization before they can use the MPI tool information interface.  $^{41}$ 

Advice to users. If MPI\_T\_INIT\_THREAD is called before MPI\_INIT\_THREAD, the 43 requested and provided thread level for MPI\_T\_INIT\_THREAD may influence the be-44havior and return value of MPI\_INIT\_THREAD. The same is true for the reverse order. 45Likewise, when using the Sessions Model (Section 11.3), the requested and provided 46thread level for MPI\_T\_INIT\_THREAD may influence the behavior and return values 47

of MPI\_SESSION\_INIT (see Section 11.3), with the same being true for the reverse order. (*End of advice to users.*)

Advice to implementors. MPI implementations should strive to make as many control or performance variables available before MPI initialization (instead of adding them during initialization) to allow tools the most flexibility. In particular, control variables should be available before MPI initialization if their value cannot be changed after MPI initialization. (*End of advice to implementors.*)

### MPI\_T\_FINALIZE()

## C binding

int MPI\_T\_finalize(void)

This routine finalizes the use of the MPI tool information interface and may be called as often as the corresponding MPI\_T\_INIT\_THREAD routine up to the current point of execution. Calling it more times returns a corresponding error code. As long as the number of calls to MPI\_T\_FINALIZE is smaller than the number of calls to MPI\_T\_INIT\_THREAD up to the current point of execution, the MPI tool information interface remains initialized and calls to its routines are permissible. Further, additional calls to MPI\_T\_INIT\_THREAD after one or more calls to MPI\_T\_FINALIZE are permissible.

Once MPI\_T\_FINALIZE is called the same number of times as the routine MPI\_T\_INIT\_THREAD up to the current point of execution, the MPI tool information interface is no longer initialized. The user can reinitialize the interface by a subsequent call to MPI\_T\_INIT\_THREAD.

At the end of the program execution, unless MPI\_ABORT is called, an application must have called MPI\_T\_INIT\_THREAD and MPI\_T\_FINALIZE an equal number of times.

### 15.3.5 Datatype System

All variables managed through the MPI tool information interface represent their values through typed buffers of a given length and type using an MPI datatype (similar to regular send/receive buffers). Since the initialization of the MPI tool information interface is separate from the initialization of MPI, MPI tool information interface routines can be called before MPI initialization. Consequently, these routines can also use MPI datatypes before MPI initialization. Therefore, within the context of the MPI tool information interface, it is permissible to use a subset of MPI datatypes as specified below before MPI initialization.

*Rationale.* The MPI tool information interface relies mainly on unsigned datatypes for integer values since most variables are expected to represent counters or resource sizes. MPI\_INT is provided for additional flexibility and is expected to be used mainly for control variables and enumeration types (see below).

Providing all basic datatypes, in particular providing all signed and unsigned variants of integer types, would lead to a larger number of types, which tools need to interpret. This would cause unnecessary complexity in the implementation of tools based on the MPI tool information interface. (*End of rationale.*)

 $\overline{7}$ 

 $^{24}$ 

	(50 CHAPTER 15. TOOL SUPPORT
1	MPI_INT
2	MPI_INT32_T
3	MPI_INT64_T
4	MPI_UNSIGNED
5	MPI_UNSIGNED_LONG
6	MPI_UNSIGNED_LONG_LONG
7	MPI_UINT32_T
8	MPI_UINT64_T
9	MPI_COUNT
10	MPI_CHAR
11	MPI_DOUBLE
12	
13	
14	Table 15.3: MPI datatypes that can be used by the MPI tool information interface
15	
16	The MPI tool information interface only relies on a subset of the basic MPI datatypes
17	and does not use any derived MPI datatypes. Table 15.3 lists all MPI datatypes that can
18	be returned by the MPI tool information interface to represent its variables.
19	The use of the datatype MPI_CHAR in the MPI tool information interface implies a null-
20	terminated character array, i.e., a string in the C language. If a variable has type MPI_CHAR,
20	the value of the count parameter returned by MPI_T_CVAR_HANDLE_ALLOC and
21	MPI_T_PVAR_HANDLE_ALLOC must be large enough to include any valid value, including
23	its terminating null character. The contents of returned MPI_CHAR arrays are only defined from index 0 through the location of the first null character.
24	from index 0 through the location of the first null character.
25	Rationale. The MPI tool information interface requires a significantly simpler type
26	<i>Rationale.</i> The MPI tool information interface requires a significantly simpler type system than MPI itself. Therefore, only its required subset must be present before
27	
28	MPI initialization and MPI implementations do not need to initialize the complete
29	MPI datatype system. (End of rationale.)
30	For variables of type MPI_INT, an MPI implementation can provide additional informa-
31	tion by associating names with a fixed number of values. We refer to this information in
32	v 0
33	the following as an enumeration. In this case, the respective calls that provide additional
34	metadata for each control or performance variable, i.e., MPI_T_CVAR_GET_INFO (Sec-
35	tion 15.3.6), MPI_T_PVAR_GET_INFO (Section 15.3.7), and MPI_T_EVENT_GET_INFO
36	(Section 15.3.8), return a handle of type MPI_T_enum that can be passed to the follow-
37	ing functions to extract additional information. Thus, the MPI implementation can de-
38	scribe variables with a fixed set of values that each represents a particular state. Each
39	enumeration type can have $N$ different values, with a fixed $N$ that can be queried using
40	MPI_T_ENUM_GET_INFO.
41	
42	
43	
44	
45	
46	
47	

 		in, name, name_ien)	
IN	enumtype	enumeration to be queried (handle)	2
	chuntype	enumeration to be queried (nanule)	3
OUT	num	number of discrete values represented by this	4
		enumeration (integer)	5
OUT	name	buffer to return the string containing the name of the	6
		enumeration item (string)	7
			8
INOUT	name_len	length of the string and/or buffer for name (integer)	9
			10

MPI\_T\_ENUM\_GET\_INFO(enumtype, num, name, name\_len)

### C binding

If enumtype is a valid enumeration, this routine returns the number of items represented by this enumeration type as well as its name. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments name and name\_len are used to return the name of the enumeration as described in Section 15.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for enumerations that the MPI implementation uses.

Names associated with individual values in each enumeration enumtype can be queried using MPI\_T\_ENUM\_GET\_ITEM.

MPI\_T\_ENUM\_GET\_ITEM(enumtype, index, value, name, name\_len)

IN	enumtype	enumeration to be queried (handle)	26
IN	index	number of the value to be queried in this enumeration (integer)	27 28
OUT	value	variable value (integer)	29 30
OUT	name	buffer to return the string containing the name of the	31 32
INOUT	and the	enumeration item (string)	32 33
INOUT	name_len	length of the string and/or buffer for name (integer)	34

### C binding

The arguments name and name\_len are used to return the name of the enumeration item as described in Section 15.3.3.

If completed successfully, the routine returns the name/value pair that describes the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

# 15.3.6 Control Variables

2 The routines described in this section of the MPI tool information interface specification 3 focus on the ability to list, query, and possibly set control variables exposed by the MPI 4 implementation. These variables can typically be used by the user to fine tune properties 5and configuration settings of the MPI implementation. On many systems, such variables 6 can be set using environment variables, although other configuration mechanisms may be 7 available, such as configuration files or central configuration registries. A typical example 8 that is available in several existing MPI implementations is the ability to specify an "eager 9 limit," i.e., an upper bound on the size of messages sent or received using an eager protocol. 10

11 11

1

# Control Variable Query Functions

<sup>13</sup> An MPI implementation exports a set of N control variables through the MPI tool infor-<sup>14</sup> mation interface. If N is zero, then the MPI implementation does not export any control <sup>15</sup> variables, otherwise the provided control variables are indexed from 0 to N-1. This index <sup>16</sup> number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of control variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (*End of advice to users.*)

The following function can be used to query the number of control variables, num\_cvar:

28 29 30

31 32

33 34

35

36

37

 $^{24}$ 

25

26

27

MPI\_T\_CVAR\_GET\_NUM(num\_cvar)

OUT num\_cvar

returns number of control variables (integer)

C binding

```
int MPI_T_cvar_get_num(int *num_cvar)
```

The function MPI\_T\_CVAR\_GET\_INFO provides access to additional information for each variable.

	desc_len, bind, se	cope)
IN	cvar_index	index of the control variable to be queried, value between 0 and $num\_cvar - 1$ (integer)
OUT	name	buffer to return the string containing the name of the control variable (string)
INOUT	name_len	length of the string and/or buffer for name (integer)
OUT	verbosity	verbosity level of this variable (integer)
OUT	datatype	MPI datatype of the information stored in the control variable (handle)
OUT	enumtype	optional descriptor for enumeration information (handle)
OUT	desc	buffer to return the string containing a description of the control variable (string)
INOUT	desc_len	length of the string and/or buffer for $desc\xspace$ (integer)
OUT	bind	type of MPI object to which this variable must be bound (integer)
OUT	scope	scope of when changes to this variable are possible
bindin nt MPI_'		<pre>(integer) ; cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype,</pre>
nt MPI_'	<pre>L_cvar_get_info(int</pre>	c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope)
After After	L_cvar_get_info(int int *verbosity char *desc, in a successful call to MP is routine that query i	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same</pre>
After After Ils to th formatic If any	<pre>I_cvar_get_info(int</pre>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa-</pre>
After After Ils to th formatic If any on will ig	<pre>I_cvar_get_info(int</pre>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same cation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter.</pre>
After After Ils to th formatio If any on will in The a	<pre>I_cvar_get_info(int</pre>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa-</pre>
After After Ils to th formatic If any on will ig The a describ If cor	<pre>I_cvar_get_info(int</pre>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length</pre>
After After Ils to th formatio If any on will is The a describ If cor ne. The	<pre>I_cvar_get_info(int</pre>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same cation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable</pre>
After After Ils to th formatic If any on will is The a describ If cor ne. The Y the MF	I_cvar_get_info(int int *verbosity char *desc, in a successful call to MP is routine that query i on. An MPI implement y OUT parameter to MI gnore the parameter and rguments name and na ed in Section 15.3.3. npleted successfully, th name must be unique PI implementation.	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) M_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used</pre>
After After Ils to th formatic If any on will ig The a describ If cor ne. The y the MF The a	I_cvar_get_info(int int *verbosity char *desc, in a successful call to MP is routine that query i on. An MPI implement v OUT parameter to MI gnore the parameter at arguments name and na ed in Section 15.3.3. npleted successfully, the name must be unique PI implementation. argument verbosity returns argument verbosity argument verbosity returns argument verbosity argument verbosity returns argument verbosity argument verbosity returns argument verbosity argument verbosi	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used urns the verbosity level of the variable (see Section 15.3.1).</pre>
After After Ils to th formatic If any on will ig The a describ If cor ne. The y the MF The a	I_cvar_get_info(int int *verbosity char *desc, in a successful call to MP is routine that query i on. An MPI implement v OUT parameter to MI gnore the parameter at arguments name and na ed in Section 15.3.3. npleted successfully, the name must be unique PI implementation. argument verbosity returns argument verbosity argument verbosity returns argument verbosity argument verbosity returns argument verbosity argument verbosity returns argument verbosity argument verbosi	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) M_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used</pre>
After After Ils to th formatic If any on will ig The a describ If cor ne. The the MF The a The a uriable. If the	<ul> <li>I_cvar_get_info(int int *verbosity char *desc, in a successful call to MP is routine that query i on. An MPI implement of OUT parameter to MI gnore the parameter and rguments name and na ed in Section 15.3.3.</li> <li>Inpleted successfully, the name must be unique PI implementation.</li> <li>Implementation.</li> <li>Implement verbosity returned argument datatype returned e variable is of type MI</li> </ul>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) PI_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used urns the verbosity level of the variable (see Section 15.3.1). urns the MPI datatype that is used to represent the control PI_INT, MPI can optionally specify an enumeration for the</pre>
After After Ils to th formatio If any on will is The a describ If cor ne. The The a The a the A The a the a the a the a the a	<ul> <li>Int sverbosity char sdesc, in a successful call to MP is routine that query i on. An MPI implement of OUT parameter to MI gnore the parameter as arguments name and na ed in Section 15.3.3.</li> <li>Inpleted successfully, the name must be unique PI implementation.</li> <li>Argument verbosity returns argument datatype returns the variable is of type MI resented by this variable is variable.</li> </ul>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) M_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used urns the verbosity level of the variable (see Section 15.3.1). urns the MPI datatype that is used to represent the control PI_INT, MPI can optionally specify an enumeration for the oble and return it in enumtype. In this case, MPI returns an</pre>
After After Ils to th formatic If any on will ig The a describ If cor he. The a The a The a ariable. If the lues rep	<ul> <li>Int *verbosity char *desc, in a successful call to MP is routine that query i on. An MPI implement of OUT parameter to MI gnore the parameter to MI gnore the parameter and na ed in Section 15.3.3.</li> <li>Inpleted successfully, the name must be unique PI implementation.</li> <li>Argument verbosity returns argument datatype returns every service of the successful of the section for the parameter of the parameter of the parameter of the parameter and name must be unique PI implementation.</li> </ul>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) M_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used urns the verbosity level of the variable (see Section 15.3.1). urns the MPI datatype that is used to represent the control PI_INT, MPI can optionally specify an enumeration for the ole and return it in enumtype. In this case, MPI returns an n then be used to gather more information as described in</pre>
After After Ils to th formatic If any on will ig The a describ If cor ne. The the MF The a The a riable. If the lues rep	<ul> <li>I_cvar_get_info(int int *verbosity char *desc, in a successful call to MP is routine that query i on. An MPI implement of OUT parameter to MI gnore the parameter to MI gnore the parameter and raguments name and na ed in Section 15.3.3.</li> <li>npleted successfully, the name must be unique PI implementation.</li> <li>argument verbosity returns argument datatype returns evariable is of type MI resented by this variable on identifier, which ca 5.3.5. Otherwise, enumer</li> </ul>	<pre>c cvar_index, char *name, int *name_len, y, MPI_Datatype *datatype, MPI_T_enum *enumtype, nt *desc_len, int *bind, int *scope) M_T_CVAR_GET_INFO for a particular variable, subsequent information about the same variable must return the same sation is not allowed to alter any of the returned values. PI_T_CVAR_GET_INFO is a NULL pointer, the implementa- nd not return a value for the parameter. ame_len are used to return the name of the control variable he routine is required to return a name of at least length with respect to all other names for control variables used urns the verbosity level of the variable (see Section 15.3.1). urns the MPI datatype that is used to represent the control PI_INT, MPI can optionally specify an enumeration for the oble and return it in enumtype. In this case, MPI returns an</pre>

Returning a description is optional. If an MPI implementation does not return a de scription, the first character for desc must be set to the null character and desc\_len must
 be set to one at the return of this call.

<sup>4</sup> The parameter bind returns the type of the MPI object to which the variable must be <sup>5</sup> bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 15.3.2).

6 The scope of a variable determines whether changing a variable's value is either local  $\overline{7}$ to the MPI process or must be done by the user across multiple connected MPI processes. 8 The latter is further split into variables that require changes in a group of MPI processes 9 and those that require collective changes among all connected MPI processes. Both cases 10can require variables on all participating MPI processes either to be set to consistent (but  $^{11}$ potentially different) values or to equal values. The description provided with the variable 12must contain an explanation about the requirements and/or restrictions for setting the 13particular variable.

On successful return from MPI\_T\_CVAR\_GET\_INFO, the argument scope will be set to
 one of the constants listed in Table 15.4.

<sup>16</sup> If the name of a control variable is equivalent across connected MPI processes, the
 <sup>17</sup> following OUT parameters must be identical: verbosity, datatype, enumtype, bind, and scope.
 <sup>18</sup> The returned description must be equivalent.

L ~ .	cope Constant	Description
M	IPI_T_SCOPE_CONSTANT	read-only, value is constant
M	IPI_T_SCOPE_READONLY	read-only, cannot be written, but can change
M	IPI_T_SCOPE_LOCAL	may be writeable, writing is a local operation
M	IPI_T_SCOPE_GROUP	may be writeable, must be set to consistent values
		across a group of connected MPI processes
M	IPI_T_SCOPE_GROUP_EQ	may be writeable, must be set to the same value
		across a group of connected MPI processes
M	IPI_T_SCOPE_ALL	may be writeable, must be set to consistent values
		across all connected MPI processes
M	IPI_T_SCOPE_ALL_EQ	may be writeable, must be set to the same value
		across all connected MPI processes
	Table 1	5.4: Scopes for control variables
cł		cope of a variable only indicates if a variable might rantee that it can be changed at any time. ( <i>End of ac</i>
cł	nangeable; it is not a guar	
cł	nangeable; it is not a guar	
${ m cl} t c$	nangeable; it is not a guar	cantee that it can be changed at any time. (End of ac
${ m cl} t c$	hangeable; it is not a guar b users.)	cantee that it can be changed at any time. (End of ac
ch ta MPI_T_	nangeable; it is not a guar o <i>users.</i> ) _CVAR_GET_INDEX(name	e, cvar_index)
ch to MPI_T_ IN	nangeable; it is not a guar o <i>users.</i> ) _CVAR_GET_INDEX(name name cvar_index	e, cvar_index) name of the control variable (string)

MPI\_T\_CVAR\_GET\_INDEX is a function for retrieving the index of a control variable given a known variable name. The name parameter is provided by the caller, and cvar\_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI\_SUCCESS on success and returns MPI\_T\_ERR\_INVALID\_NAME if name does not match the name of any control variable provided by the implementation at the time of the call.

*Rationale.* This routine is provided to enable fast retrieval of control variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (*End of rationale.*)

Example 15.4 Querying and printing the names of all available control variables.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
  int i, err, num, namelen, bind, verbose, scope;
  int threadsupport;
  char name[100];
  MPI_Datatype datatype;
  err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport);
  if (err!=MPI_SUCCESS)
    return err;
  err=MPI_T_cvar_get_num(&num);
  if (err!=MPI_SUCCESS)
    return err;
  for (i=0; i<num; i++) {</pre>
    namelen=100;
    err=MPI_T_cvar_get_info(i, name, &namelen,
            &verbose, &datatype, NULL,
            NULL, NULL, /*no description */
            &bind, &scope);
    if (err!=MPI_SUCCESS && err!=MPI_T_ERR_INVALID_INDEX) return err;
    printf("Var %i: %s\n", i, name);
  }
  err=MPI_T_finalize();
```

742		CHAPTER 15. TOOL SUPPORT
if (er	r!=MPI_SUCCESS)	
	ırn 1;	
else		
	ırn 0;	
}		
Handle Al	location and Deallocatic	on
	o 0	lue of a variable, a user must first allocate a handle of type le by binding it to an MPI object (see also Section $15.3.2$ ).
han befo part	dles used in the remain ore MPI is initialized a	d in the MPI tool information interface are distinct from ing parts of the MPI standard because they must be usable and after MPI is finalized. Further, accessing handles, in variables, can be time critical and having a separate handle as. ( <i>End of rationale.</i> )
MPI_T_C	VAR_HANDLE_ALLOC	(cvar_index, obj_handle, handle, count)
IN	cvar_index	index of control variable for which handle is to be allocated (index)
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)
OUT	handle	allocated handle (handle)
OUT	count	number of elements used to represent this variable (integer)
C bindir	~~~	
	T_cvar_handle_alloc	:(int cvar_index, void *obj_handle, ndle *handle, int *count)
The object the object call for the handle all ful return	t is passed in the argun t's handle. The argum his control variable retu located to reference the h, count contains the nu	ol variable specified by the argument index to an MPI object. ment obj_handle as an address to a local variable that stores ment obj_handle is ignored if the MPI_T_CVAR_GET_INFO urned MPI_T_BIND_NO_OBJECT in the argument bind. The variable is returned in the argument handle. Upon success- umber of elements (of the datatype returned by a previous used to represent this variable.
cont	trol variable was bound	Int can be different based on the MPI object to which the d. For example, variables bound to communicators could a the size of the communicator.
MPI libra add	_COMM_WORLD to this ary. Instead, such obj	references to predefined MPI object handles, such as s routine, since their implementation depends on the MPI ject handles should be stored in a local variable and the ole should be passed into MPI_T_CVAR_HANDLE_ALLOC.

1 The value of cvar\_index should be in the range from 0 to  $num_cvar - 1$ , where  $num_cvar$ 2 is the number of available control variables as determined from a prior call to 3 MPI\_T\_CVAR\_GET\_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI\_T\_CVAR\_GET\_INFO. 4 56 MPI\_T\_CVAR\_HANDLE\_FREE(handle) 7 8 INOUT handle handle to be freed (handle) 9 10 C binding 11 int MPI\_T\_cvar\_handle\_free(MPI\_T\_cvar\_handle \*handle) 12When a handle is no longer needed, a user of the MPI tool information interface should 13 call MPI\_T\_CVAR\_HANDLE\_FREE to free the handle and the associated resources in the 14MPI implementation. On a successful return, MPI sets the handle to 15MPI\_T\_CVAR\_HANDLE\_NULL. 1617 Control Variable Access Functions 18 19 2021MPI\_T\_CVAR\_READ(handle, buf) 22 IN handle handle to the control variable to be read (handle) 23OUT buf initial address of storage location for variable value  $^{24}$ (choice) 2526C binding 27int MPI\_T\_cvar\_read(MPI\_T\_cvar\_handle handle, void \*buf) 28 29 This routine queries the value of a control variable identified by the argument handle and 30 stores the result in the buffer identified by the parameter **buf**. The user must ensure that the 31buffer is of the appropriate size to hold the entire value of the control variable (based on the 32 returned datatype and count from prior corresponding calls to MPI\_T\_CVAR\_GET\_INFO 33 and MPI\_T\_CVAR\_HANDLE\_ALLOC, respectively). 34 3536 MPI\_T\_CVAR\_WRITE(handle, buf) 37 IN handle handle to the control variable to be written (handle) 38 IN buf initial address of storage location for variable value 39 (choice) 40 41 42C binding 43 int MPI\_T\_cvar\_write(MPI\_T\_cvar\_handle handle, const void \*buf) 44This routine sets the value of the control variable identified by the argument handle to 45

This routine sets the value of the control variable identified by the argument handle to the data stored in the buffer identified by the parameter **buf**. The user must ensure that the buffer is of the appropriate size to hold the entire value of the control variable (based on the

47 48

1	returned datatype and count from prior corresponding calls to MPI_T_CVAR_GET_INFO			
2	and MPI_T_CVAR_HANDLE_ALLOC, respectively).			
3	If the variable has a global scope (as returned by a prior corresponding			
4	$MPI\_T\_CVAR\_GET\_INFO$ call), any write call to this variable must be issued by the user			
5	in all connected (as defined in Section $11.10.4$ ) MPI processes. If the variable has group			
6	scope, any write call to this variable must be issued by the user in all MPI processes in			
7	the group, which must be described by the $MPI$ implementation in the description by the			
8	MPI_T_CVAR_GET_INFO.			
9 10	In both cases, the user must ensure that the writes in all participating MPI processes are consistent. If the scope is either MPI_T_SCOPE_ALL_EQ or MPI_T_SCOPE_GROUP_EQ			
11 12	this means that the variable in all connected MPI processes or MPI processes of the group, respectively, must be set to the same value.			
13	If it is not possible to change the variable at the time the call is made, the function			
14	returns either MPI_T_ERR_CVAR_SET_NOT_NOW, if there may be a later time at which the			
15	variable could be set, or MPI_T_ERR_CVAR_SET_NEVER, if the variable cannot be set for the			
16	remainder of the application's execution.			
17				
18	<b>Example 15.5</b> Reading the value of a control variable.			
19	int actualus int comm (int index MDT Comm comm int wwol) (			
20	<pre>int getValue_int_comm(int index, MPI_Comm comm, int *val) {     int orp count;</pre>			
21	<pre>int err,count; MPI_T_cvar_handle handle;</pre>			
22	MF1_1_CVa1_Handle Handle,			
23	/* This example assumes that the variable index */			
24	/* can be bound to a communicator */			
25				
26	err=MPI_T_cvar_handle_alloc(index, &comm, &handle, &count);			
27	if (err!=MPI_SUCCESS) return err;			
28				
29	/* The following assumes that the variable is */			
30	/* represented by a single integer */			
31	,,,,,,			
32	err=MPI_T_cvar_read(handle,val);			
33	if (err!=MPI_SUCCESS) return err;			
34	(			
35	err=MPI_T_cvar_handle_free(&handle);			
36 27	return err;			
37	}			
38				
39	15.3.7 Performance Variables			

40

# 15.3.7 Performance Variables

41The following section focuses on the ability to list and to query performance variables 42provided by the MPI implementation. Performance variables provide insight into MPI 43implementation-specific internals and can represent information such as the state of the 44MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data 45for submodules, or queue sizes and lengths. 46

47Rationale. The interface for performance variables is separate from the interface for 48 control variables, since performance variables have different requirements and parameters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (*End of rationale.*)

Some performance variables and classes refer to *events*. In general, such events describe state transitions within software or hardware related to the performance of an MPI application. The events offered through the callback-driven event-notification interface described in Section 15.3.8 also refer to such state transitions; however, the set of state transitions referred to by performance variables and events as described in Section 15.3.8 may not be identical.

### Performance Variable Classes

Each performance variable is associated with a class that describes its basic semantics, possible datatypes, basic behavior, its starting value, whether it can overflow, and when and how an MPI implementation can change the variable's value. The starting value is the value that is assigned to the variable the first time that it is used or whenever it is reset.

Advice to users. If a performance variable belongs to a class that can overflow, it is up to the user to protect against this overflow, e.g., by frequently reading and resetting the variable value. (*End of advice to users.*)

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (*End* of advice to implementors.)

The classes are defined by the following constants:

# • MPI\_T\_PVAR\_CLASS\_STATE

A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI\_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 15.3.5. The starting value is the current state of the implementation at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

### • MPI\_T\_PVAR\_CLASS\_LEVEL

A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

• MPI\_T\_PVAR\_CLASS\_SIZE

A performance variable in this class represents a value that is the size of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG,

MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current size of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow. 

 $\mathbf{5}$ 

- 1 MPI\_T\_PVAR\_CLASS\_PERCENTAGE 2 The value of a performance variable in this class represents the percentage utiliza-3 tion of a finite resource. The value of a variable of this class can change at any 4 time to match the current utilization level of the resource. It will be returned as an 5MPI\_DOUBLE datatype. The value must always be between 0.0 (resource not used at 6 all) and 1.0 (resource completely used). The starting value is the current percent-7 age utilization level of the resource at the time that the starting value is set. MPI 8 implementations must ensure that variables of this class cannot overflow. 9 MPI\_T\_PVAR\_CLASS\_HIGHWATERMARK 10 A performance variable in this class represents a value that describes the high water-11 mark utilization of a resource. The value of a variable of this class is non-negative 12and grows monotonically from the initialization or reset of the variable. It can be rep-13 resented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, 14MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utiliza-15tion level of the resource at the time that the variable is started or reset. MPI imple-16mentations must ensure that variables of this class cannot overflow. 17 18 MPI\_T\_PVAR\_CLASS\_LOWWATERMARK 19 A performance variable in this class represents a value that describes the low water-20mark utilization of a resource. The value of a variable of this class is non-negative 21and decreases monotonically from the initialization or reset of the variable. It can be 22 represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, 23MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value is the current utiliza-24tion level of the resource at the time that the variable is started or reset. MPI imple-25mentations must ensure that variables of this class cannot overflow. 26 MPI\_T\_PVAR\_CLASS\_COUNTER 27A performance variable in this class counts the number of occurrences of a specific 28 event (e.g., the number of memory allocations within an MPI library). The value of 29 a variable of this class increases monotonically from the initialization or reset of the 30 performance variable by one for each specific event that is observed. Values must 31be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, 32 MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG. The starting value for variables 33 of this class is 0. Variables of this class can overflow. 34 35 MPI\_T\_PVAR\_CLASS\_AGGREGATE 36 The value of a performance variable in this class is an an aggregated value that 37 represents a sum of arguments processed during a specific event (e.g., the amount 38 of memory allocated by all memory allocations). This class is similar to the counter 39 class, but instead of counting individual events, the value can be incremented by 40 arbitrary amounts. The value of a variable of this class increases monotonically from 41 the initialization or reset of the performance variable. It must be non-negative and 42represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, 43 MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value for variables of this 44class is 0. Variables of this class can overflow. 45
  - MPI\_T\_PVAR\_CLASS\_TIMER

46

47

48

The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event, type of event, or section

of the MPI library. This class has the same basic semantics as MPI\_T\_PVAR\_CLASS\_AGGREGATE, but explicitly records a timing value. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_UNSIGNED\_LONG\_LONG, MPI\_DOUBLE. The starting value for variables of this class is 0. If the type MPI\_DOUBLE is used, the units that represent time in this datatype must match the units used by MPI\_WTIME. Otherwise, the time units should be documented, e.g., in the description returned by MPI\_T\_PVAR\_GET\_INFO. Variables of this class can overflow.

• MPI\_T\_PVAR\_CLASS\_GENERIC

This class can be used to describe a variable that does not fit into any of the other classes. For variables in this class, the starting value is variable-specific and implementation-defined.

### Performance Variable Query Functions

An MPI implementation exports a set of N performance variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any performance variables; otherwise the provided performance variables are indexed from 0 to N-1. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, num\_pvar:

MPI_T_PVAR_GET_NUM(num_pvar)	

num\_pvar

returns number of performance variables (integer)

### C binding

OUT

int MPI\_T\_pvar\_get\_num(int \*num\_pvar)

The function MPI\_T\_PVAR\_GET\_INFO provides access to additional information for each variable.

1

 $\mathbf{2}$ 

3

4

5 6

 $\overline{7}$ 

8

9

10 11

12

13

14

15 16

17 18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

34 35

36

37 38

39

1		(AD CET INFO(much index a	ana nama lan washasita was alaga datatuna		
2			ame, name_len, verbosity, var_class, datatype, n, bind, readonly, continuous, atomic)		
3 4 5	IN	pvar_index	index of the performance variable to be queried between 0 and $num\_pvar - 1$ (integer)		
6 7	OUT	name	buffer to return the string containing the name of the performance variable (string)		
8	INOUT	name_len	length of the string and/or buffer for name (integer) $% \left( {{\left[ {{{\left[ {{{\left[ {{\left[ {{\left[ {{\left[ {{{\left[ {{{\left[ {{{\left[ {{\left[ {{{\left[ {{{\left[ {{{\left[ {{{\left[ {{{}}}} \right]}}}} \right.$		
9 10	OUT	verbosity	verbosity level of this variable (integer)		
11	OUT	var_class	class of performance variable (integer)		
12 13	OUT	datatype	MPI data type of the information stored in the performance variable (handle)		
14 15	OUT	enumtype	optional descriptor for enumeration information (handle)		
16 17 18	OUT	desc	buffer to return the string containing a description of the performance variable (string)		
19	INOUT	desc_len	length of the string and/or buffer for desc (integer)		
20 21	OUT	bind	type of MPI object to which this variable must be bound (integer)		
22 23 24	OUT	readonly	flag indicating whether the variable can be written/reset (integer)		
25 26	OUT	continuous	flag indicating whether the variable can be started and stopped or is continuously active (integer)		
27 28	OUT	atomic	flag indicating whether the variable can be atomically read and reset (integer)		
29 30					
31	C binding		index, char *name, int *name_len,		
32 33	1		<pre>*var_class, MPI_Datatype *datatype,</pre>		
34 35		MPI_T_enum *enumtype	, char *desc, int *desc_len, int *bind, continuous, int *atomic)		
35 36 37		a successful call to MPI_T_PV	AR_GET_INFO for a particular variable, subsequent		
38		calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values.			
39	If any	If any OUT parameter to MPI_T_PVAR_GET_INFO is a NULL pointer, the implementa-			
40	-	tion will ignore the parameter and not return a value for the parameter.			
41 42		The arguments name and name_len are used to return the name of the performance variable as described in Section 15.3.3. If completed successfully, the routine is required			
43		a name of at least length one.	If completed successfully, the routine is required		
44		0	verbosity level of the variable (see Section 15.3.1).		
45	The class of the performance variable is returned in the parameter var_class. The class				
46	must be one of the constants defined in Section 15.3.7.				
47 48	The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.				

Advice to implementors. Groups of variables that belong closely together, but have different classes, can have the same name. This choice is useful, e.g., to refer to multiple variables that describe a single resource (like the level, the total size, as well as high and low watermarks). (End of advice to implementors.)

The argument datatype returns the MPI datatype that is used to represent the performance variable.

If the variable is of type MPI\_INT, MPI can optionally specify an enumeration for the values represented by this variable and return it in enumtype. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 15.3.5. Otherwise, enumtype is set to MPI\_T\_ENUM\_NULL. If the datatype is not MPI\_INT or the argument enumtype is the null pointer, no enumeration type is returned.

Returning a description is optional. If an MPI implementation does not return a description, the first character for desc must be set to the null character and desc\_len must be set to one at the return from this function.

The parameter bind returns the type of the MPI object to which the variable must be bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 15.3.2).

Upon return, the argument **readonly** is set to zero if the variable can be written or reset by the user. It is set to one if the variable can only be read.

Upon return, the argument **continuous** is set to zero if the variable can be started and stopped by the user, i.e., it is possible for the user to control if and when the value of a variable is updated. It is set to one if the variable is always active and cannot be controlled by the user.

Upon return, the argument **atomic** is set to zero if the variable cannot be read and reset atomically. Only variables for which the call sets **atomic** to one can be used in a call to MPI\_T\_PVAR\_READRESET.

If a performance variable has an equivalent name and has the same class across connected MPI processes, the following OUT parameters must be identical: verbosity, varclass, datatype, enumtype, bind, readonly, continuous, and atomic. The returned description must be equivalent.

MPI\_T\_PVAR\_GET\_INDEX(name, var\_class, pvar\_index)

IN	name	the name of the performance variable (string)
IN	var_class	the class of the performance variable (integer)
OUT	pvar_index	the index of the performance variable (integer)

### C binding

int MPI\_T\_pvar\_get\_index(const char \*name, int var\_class, int \*pvar\_index)

MPI\_T\_PVAR\_GET\_INDEX is a function for retrieving the index of a performance variable given a known variable name and class. The name and var\_class parameters are provided by the caller, and pvar\_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI\_SUCCESS on success and returns MPI\_T\_ERR\_INVALID\_NAME if name does not match the name of any performance variable of the specified var\_class provided by the implementation at the time of the call.

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

Rationale. This routine is provided to enable fast retrieval of performance variables by a tool, assuming it knows the name of the variable for which it is looking. The number of variables exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of variables once at initialization. Although using MPI implementation specific variable names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of variables to find a specific one. (End of rationale.)

# <sup>10</sup> Performance Experiment Sessions

Within a single program, multiple components can use the MPI tool information interface. To avoid collisions with respect to accesses to performance variables, users of the MPI tool information interface must first create a performance experiment session. Subsequent calls that access performance variables can then be made within the context of this performance experiment session. Starting, stopping, reading, writing, or resetting a variable in one performance experiment session shall not influence whether a variable is started, stopped, read, written, or reset in another performance experiment session.

```
19
20
```

21

22 23

25 26

27

28

```
MPI_T_PVAR_SESSION_CREATE(pe_session)
```

identifier of performance experiment session (handle)

# <sup>24</sup> C binding

OUT

```
int MPI_T_pvar_session_create(MPI_T_pvar_session *pe_session)
```

This call creates a new performance experiment session for accessing performance variables and returns a handle for this performance experiment session in the argument pe\_session of type MPI\_T\_pvar\_session.

```
29
30
31
```

32

33 34

36 37

38

39

```
MPI_T_PVAR_SESSION_FREE(pe_session)
```

pe\_session

INOUT pe\_session

identifier of performance experiment session (handle)

# 35 C binding

```
int MPI_T_pvar_session_free(MPI_T_pvar_session *pe_session)
```

This call frees an existing performance experiment session. Calls to the MPI tool information interface can no longer be made within the context of a performance experiment session after it is freed. On a successful return, MPI sets the performance experiment session identifier to MPI\_T\_PVAR\_SESSION\_NULL.

40 41 42

43

```
Handle Allocation and Deallocation
```

Before using a performance variable, a user must first allocate a handle of type
 MPI\_T\_pvar\_handle for the variable by binding it to an MPI object (see also Section 15.3.2).
 46
 47
 48

1

2

3

4

5

6

7

8

•••			
	IN	pe_session	identifier of performance experiment session (handle)
	IN	pvar_index	index of performance variable for which handle is to be allocated (integer)
	IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound (pointer)
	OUT	handle	allocated handle (handle)
	OUT	count	number of elements used to represent this variable (integer)

MPI\_T\_PVAR\_HANDLE\_ALLOC(pe\_session, pvar\_index, obj\_handle, handle, count)

## C binding

This routine binds the performance variable specified by the argument index to an MPI object in the performance experiment session identified by the parameter pe\_session. The object is passed in the argument obj\_handle as an address to a local variable that stores the object's handle. The argument obj\_handle is ignored if the MPI\_T\_PVAR\_GET\_INFO call for this performance variable returned MPI\_T\_BIND\_NO\_OBJECT in the argument bind. The handle allocated to reference the variable is returned in the argument handle. Upon successful return, count contains the number of elements (of the datatype returned by a previous MPI\_T\_PVAR\_GET\_INFO call) used to represent this variable.

Advice to users. The count can be different based on the MPI object to which the performance variable was bound. For example, variables bound to communicators could have a count that matches the size of the communicator.

It is not portable to pass references to predefined MPI object handles, such as MPI\_COMM\_WORLD, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into MPI\_T\_PVAR\_HANDLE\_ALLOC. (*End of advice to users.*)

The value of index should be in the range from 0 to  $num_pvar - 1$ , where  $num_pvar$  is the number of available performance variables as determined from a prior call to MPI\_T\_PVAR\_GET\_NUM. The type of the MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI\_T\_PVAR\_GET\_INFO.

For all routines in the rest of this section that take both handle and pe\_session as IN or INOUT arguments, if the handle argument passed in is not associated with the pe\_session argument, MPI\_T\_ERR\_INVALID\_HANDLE is returned.

 MPI\_T\_PVAR\_HANDLE\_FREE(pe\_session, handle)

 IN
 pe\_session

 INOUT
 handle

 INOUT
 handle

 C binding

int MPI\_T\_pvar\_handle\_free(MPI\_T\_pvar\_session pe\_session,

 $^{24}$ 

 $^{31}$ 

1	MPI_T_pvar_handle *handle)				
2 3 4 5 6	When a handle is no longer needed, a user of the MPI tool information interface should call MPI_T_PVAR_HANDLE_FREE to free the handle in the performance experiment session identified by the parameter pe_session and the associated resources in the MPI implementation. On a successful return, MPI sets the handle to MPI_T_PVAR_HANDLE_NULL.				
7 8	Starting a	Starting and Stopping of Performance Variables			
9 10 11 12 13 14	Performance variables that have the continuous flag set during the query operation are continuously operating once a handle has been allocated. Such variables may be queried at any time, but they cannot be started or stopped by the user. All other variables are in a stopped state after their handle has been allocated; their values are not updated until they have been started by the user.				
15 16	MPI_T_F	VAR_START(pe_sess	on, handle)		
17	IN	pe_session	identifier of performance experiment session (he	andle)	
18 19	IN	handle	handle of a performance variable (handle)		
22 23 24 25 26 27 28 29 30 31 32 33	C binding int MPI_T_pvar_start(MPI_T_pvar_session pe_session,				
34 35	MPI T F	PVAR_STOP(pe_session	handle)		
36	IN IN	pe_session	identifier of performance experiment session (ha	andle)	
37 38 39	IN	handle	handle of a performance variable (handle)	unano)	
40 41 42 43 44 45 46 47	C binding int MPI_T_pvar_stop(MPI_T_pvar_session pe_session, MPI_T_pvar_handle handle) This functions stops the performance variable with the handle identified by the para eter handle in the performance experiment session identified by the parameter pe_session If the constant MPI_T_PVAR_ALL_HANDLES is passed in handle, the MPI implement tion attempts to stop all variables within the performance experiment session identified			sion. nenta-	
48					

by the parameter **pe\_session** for which handles have been allocated. In this case, the routine returns MPI\_SUCCESS if all variables are stopped successfully (even if there are no noncontinuous variables to be stopped), otherwise MPI\_T\_ERR\_PVAR\_NO\_STARTSTOP is returned. Continuous variables and variables that are already stopped are ignored when MPI\_T\_PVAR\_ALL\_HANDLES is specified.

# Performance Variable Access Functions

MPI\_T\_PVAR\_READ(pe\_session, handle, buf)INpe\_sessionidentifier of performance experiment session (handle)INhandlehandle of a performance variable (handle)OUTbufinitial address of storage location for variable value (choice)

# C binding

int	MPI_T_pvar_read(MPI_T_pvar_ses	sion pe_session,	,
	MPI_T_pvar_handle han	dle, void *buf)	

The MPI\_T\_PVAR\_READ call queries the value of the performance variable with the handle handle in the performance experiment session identified by the parameter pe\_session and stores the result in the buffer identified by the parameter buf. The user is responsible to ensure that the buffer is of the appropriate size to hold the entire value of the performance variable (based on the datatype and count returned by the corresponding previous calls to MPI\_T\_PVAR\_GET\_INFO and MPI\_T\_PVAR\_HANDLE\_ALLOC, respectively).

The constant MPI\_T\_PVAR\_ALL\_HANDLES cannot be used as an argument for the function MPI\_T\_PVAR\_READ.

MPI\_T\_PVAR\_WRITE(pe\_session, handle, buf)

IN	pe_session	identifier of performance experiment session (handle)	32
IN	handle	handle of a performance variable (handle)	33
		-	34
IN	buf	initial address of storage location for variable value	35
		(choice)	36

# C binding

The MPI\_T\_PVAR\_WRITE call attempts to write the value of the performance variable with the handle identified by the parameter handle in the performance experiment session identified by the parameter pe\_session. The value to be written is passed in the buffer identified by the parameter buf. The user must ensure that the buffer is of the appropriate size to hold the entire value of the performance variable (based on the datatype and count returned by the corresponding previous calls to MPI\_T\_PVAR\_GET\_INFO and MPI\_T\_PVAR\_HANDLE\_ALLOC, respectively).

 $\mathbf{2}$ 

 $^{24}$ 

1 2		is not possible to c RR_PVAR_NO_WRITE.	hange the variable, the function returns	
3	The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the func-			
4	tion $MPI$	_T_PVAR_WRITE.		
5				
6 7	MPI_T_F	VAR_RESET(pe_sess	ion, handle)	
8	IN	pe_session	identifier of performance experiment session (handle)	
9 10	IN	handle	handle of a performance variable (handle)	
11	C hindi			
12	C bindi	0	T_pvar_session pe_session,	
13	IIIC MFI	-	andle handle)	
14		-		
15 16			T call sets the performance variable with the handle identified starting value specified in Section $15.3.7$ . If it is not possible	
17	to change	e the variable, the fur	action returns MPI_T_ERR_PVAR_NO_WRITE.	
18			<code>AR_ALL_HANDLES</code> is passed in <code>handle</code> , the <code>MPI</code> implementation	
19	-		within the performance experiment session identified by the	
20	-	•	handles have been allocated. In this case, the routine returns	
21 22			re reset successfully (even if there are no valid handles or all	
23		- / ·	T_ERR_PVAR_NO_WRITE is returned. Read-only variables are	
24	ignored v	vnen MPI_I_PVAR_AL	L_HANDLES is specified.	
25				
26	MPI_T_F	PVAR_READRESET(p	e_session, handle, buf)	
27	IN	pe_session	identifier of performance experiment session (handle)	
28 29	IN	handle	handle of a performance variable (handle)	
30 31	OUT	buf	initial address of storage location for variable value (choice)	
32				
33	C bindi	0		
34 35	int MPI	-	MPI_T_pvar_session pe_session, nandle handle, void *buf)	
36	This	call atomically com	bines the functionality of MPI_T_PVAR_READ and	
37			e same semantics as if these two calls were called separately.	
38			variable are not supported, this routine returns	
39		RR_PVAR_NO_ATOMI		
40 41	The	constant MPI_T_PVA	<b>R_ALL_HANDLES</b> cannot be used as an argument for the func-	
42	tion $MPI$	_T_PVAR_READRES	ET.	
43	Ad	vice to implementors.	Sampling-based tools rely on the ability to call the MPI tool	
44			particular routines to start, stop, read, write, and reset per-	
45			any program context, including asynchronous contexts such	
46		-	implementations should strive, if possible in their particular	
47 48	environment, to enable these usage scenarios for all or a subset of the routines men- tioned above. If implementing only a subset, the read, write, and reset routines are			

# 15.3. THE MPI TOOL INFORMATION INTERFACE

typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by MPI\_T\_PVAR\_GET\_INFO. (*End of advice to implementors.*)

*Rationale.* All routines to read, to write or to reset performance variables require the performance experiement session argument. This requirement keeps the interface consistent and allows the use of MPI\_T\_PVAR\_ALL\_HANDLES where appropriate. Further, this opens up additional performance optimizations for the implementation of handles. (*End of rationale.*)

**Example 15.6** Detecting Receives with long unexpected message queues.

The following example shows a sample tool to identify receive operations that occur during times with long message queues. This examples assumes that the MPI implementation exports a variable with the name "MPI\_T\_UMQ\_LENGTH" to represent the current length of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI profiling interface.

The tool consists of three parts: (1) the initialization (by intercepting the call to MPI\_INIT), (2) the test for long unexpected message queues (by intercepting calls to MPI\_RECV), and (3) the clean-up phase (by intercepting the call to MPI\_FINALIZE). To capture all receives, the example would have to be extended to have similar wrappers for all receive operations.

Part 1—Initialization: During initialization, the tool searches for the variable and, once the right index is found, allocates a performance experiment session and a handle for the variable with the found index, and starts the performance variable.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <mpi.h>
/* Global variables for the tool */
static MPI_T_pvar_session pe_session;
static MPI_T_pvar_handle handle;
int MPI_Init(int *argc, char ***argv ) {
      int err, num, i, index, namelen, verbosity;
      int var_class, bind, threadsup;
      int readonly, continuous, atomic, count;
      char name [18];
      MPI_Comm comm;
      MPI_Datatype datatype;
      MPI_T_enum enumtype;
      err=PMPI_Init(argc, argv);
```

755

1 2

3

4

 $\mathbf{5}$ 

6

7

8

9

10

11

12 13

14

15

16

17

18

19

20

21

22

23 24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

42

43

44

45

 $46 \\ 47$ 

```
1
           if (err!=MPI_SUCCESS) return err;
2
3
           err=PMPI_T_init_thread(MPI_THREAD_SINGLE, &threadsup);
4
           if (err!=MPI_SUCCESS) return err;
5
6
           err=PMPI_T_pvar_get_num(&num);
7
           if (err!=MPI_SUCCESS) return err;
8
           index=-1;
9
           i=0;
10
           while ((i<num) && (index<0) && (err==MPI_SUCCESS)) {</pre>
11
                  /* Pass a buffer that is at least one character longer than */
12
                  /* the name of the variable being searched for to avoid */
13
                  /* finding variables that have a name that has a prefix */
14
                  /* equal to the name of the variable being searched. */
15
                  namelen=18;
16
                  err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity,
17
                          &var_class, &datatype, &enumtype, NULL, NULL, &bind,
18
                          &readonly, &continuous, &atomic);
19
                  if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i;
20
                  i++; }
21
           if (err!=MPI_SUCCESS) return err;
22
23
           /* this could be handled in a more flexible way for a generic tool */
24
           assert(index>=0);
25
           assert(var_class==MPI_T_PVAR_CLASS_LEVEL);
26
           assert(datatype==MPI_INT);
27
           assert(bind==MPI_T_BIND_MPI_COMM);
28
29
           /* Create a session */
30
           err=PMPI_T_pvar_session_create(&pe_session);
31
           if (err!=MPI_SUCCESS) return err;
32
33
           /* Get a handle and bind to MPI_COMM_WORLD */
34
           comm=MPI_COMM_WORLD;
35
           err=PMPI_T_pvar_handle_alloc(pe_session, index, &comm, &handle,
36
                                          &count);
37
           if (err!=MPI_SUCCESS) return err;
38
39
           /* this could be handled in a more flexible way for a generic tool */
40
           assert(count==1);
41
42
           /* Start variable */
43
           err=PMPI_T_pvar_start(pe_session, handle);
44
           if (err!=MPI_SUCCESS) return err;
45
46
           return MPI_SUCCESS;
47
     }
48
```

Part 2—Testing the Queue Lengths During Receives: During every receive operation, the tool reads the unexpected queue length through the matching performance variable and compares it against a predefined threshold.

```
#define THRESHOLD 5
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
{
        int value, err;
        if (comm==MPI_COMM_WORLD) {
                err=PMPI_T_pvar_read(pe_session, handle, &value);
                if ((err==MPI_SUCCESS) && (value>THRESHOLD))
                 ſ
                         /* tool identified receive called with long UMQ */
                         /* execute tool functionality, */
                         /* e.g., gather and print call stack */
                }
        }
        return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
}
Part 3—Termination: In the wrapper for MPI_FINALIZE, the MPI tool information interface
is finalized.
int MPI_Finalize(void)
{
    int err;
    err=PMPI_T_pvar_handle_free(pe_session, &handle);
    err=PMPI_T_pvar_session_free(&pe_session);
    err=PMPI_T_finalize();
```

# 15.3.8 Events

}

return PMPI\_Finalize();

During the execution of an MPI application, the MPI implementation can raise events of 39 a specific type to inform the user of a state change in the implementation. Event types 40describe specific state changes within the MPI implementation. In comparison to aggregate 41 performance variables, events provide per-instance information on such state changes. The 42MPI implementation is said to *raise an event* when it invokes a callback function previously 43 registered for the corresponding event type by the user. Each callback invocation for a 44specific event instance has a timestamp associated with it, which can be queried by the user, 45describing the time when the event was observed by the implementation. This decouples 46the observation of the state change from the communication of this information to the user. 47A timestamp in this context is a count of clock ticks elapsed since some time in the past 48

1

2

3

4 5

> 6 7

8

9

10

11 12

13

14

15

16

17

18

19

20

21 22

23

24 25

26

27 28

29

30

31

32

33

34

35

36 37

```
1
      and represented as a variable of type MPI_Count.
\mathbf{2}
3
      Event Sources
4
      As a means to manage multiple state changes to be observed concurrently by different
5
      parts of the software and hardware system, the event interface of the MPI Tool Information
6
      Interface uses the concept of sources. A source in this context is a concept describing the
7
      logical entity raising the event. A source may or may not directly represent a concrete
8
      part of the software or hardware system. This concept is used primarily to describe partial
9
      ordering of events across different components where total ordering cannot necessarily be
10
      determined or is too costly to enforce.
11
          The following function can be used to query the number of event sources, num_sources:
12
13
14
      MPI_T_SOURCE_GET_NUM(num_sources)
15
        OUT
                                                returns number of event sources (integer)
                  num_sources
16
17
      C binding
18
19
      int MPI_T_source_get_num(int *num_sources)
20
          The number of available event sources can be queried with a call to
21
      MPI_T_SOURCE_GET_NUM. An MPI implementation is allowed to increase the number of
22
      sources during the execution of an MPI process. However, MPI implementations are not
23
      allowed to change the index of an event source or to delete an event source once it has been
^{24}
      made visible to the user (e.g., if new event sources become available via dynamic loading of
25
      additional components in the MPI implementation).
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
```

MPI_T_SOURCE_GET_INFO(source_index, name, name_len, desc, desc_len, ordering, ticks_per_second, max_ticks, info) <sup>2</sup>			
IN	source_index	index of the source to be queried between 0 and num_sources $-1$ (integer)	3 4 5
OUT	name	buffer to return the string containing the name of the source (string)	6 7
INOUT	name_len	length of the string and/or buffer for name (integer)	8
OUT	desc	buffer to return the string containing the description of the source (string)	9 10 11
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)	11
OUT	ordering	flag indicating chronological ordering guarantees given by the source (integer)	13 14
OUT	ticks_per_second	the number of ticks per second for the timer of this source (integer)	15 16
OUT	max_ticks	the maximum count of ticks reported by this source before overflow occurs (integer)	17 18 19
OUT	info	optional info object (handle)	20
			21
C binding	r 5		22
int MPI_T	_source_get_info(int sour	ce_index, char *name, int *name_len,	23
		<pre>c_len, MPI_T_source_order *ordering,</pre>	24
	-	_second, MPI_Count *max_ticks,	25 26
	MPI_Info *info)		20 27
A call	to MPI_T_SOURCE_GET_IN	<b>IFO</b> returns additional information on the source	28
	by the source_index $\operatorname{argument}$ .		29
			30
	Section 15.3.3.		31
The arguments desc and desc_len are used to return the description of the source as $_{32}$			

The arguments desc and desc\_len are used to return the description of the source as described in Section 15.3.3.

The ordering argument returns whether event callbacks of this source will be invoked in chronological order, i.e., the timestamps reported by MPI\_T\_EVENT\_GET\_TIMESTAMP of subsequent events of the same source are monotonically increasing. The value of ordering can be MPI\_T\_SOURCE\_ORDERED or MPI\_T\_SOURCE\_UNORDERED.

The ticks\_per\_seconds argument returns the number of ticks elapsed in one second for the timer used for the specific source.

The max\_ticks argument returns the largest number of ticks reported by this source as a timestamp before the value overflows.

Advice to users. As the size of MPI\_Count is defined in relation to the types MPI\_Aint and MPI\_Offset, the effective size of MPI\_Count may lead to overflows of the timestamp values reported. Users can use the argument max\_ticks to mitigate resulting problems. (*End of advice to users.*)

MPI can optionally return an info object containing the default hints set for this source. If the argument to info provided by the user is the NULL pointer, this argument is ignored,

33

34

35

36

37

38

39

40

 $41 \\ 42$ 

43

44

45

 $46 \\ 47$ 

otherwise an MPI implementation is required to return all hints that are supported by  $\mathbf{2}$ the implementation for this source and have default values specified; any user-supplied 3 hints that were not ignored by the implementation; and any additional hints that were 4 set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info via 6 MPI\_INFO\_FREE.

MPI\_T\_SOURCE\_GET\_TIMESTAMP(source\_index, timestamp) 9 10 source\_index IN index of the source (integer) 11OUT timestamp current timestamp from specified source (integer) 12

# C binding

```
14
     int MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)
```

15

13

16To enable proper query of a reference timestamp for a specific source, a user can obtain 17a current timestamp using MPI\_T\_SOURCE\_GET\_TIMESTAMP. The argument 18source\_index identifies the index of the source to query. The call returns MPI\_SUCCESS and 19a current timestamp in the argument timestamp if the source supports ad-hoc generation of 20timestamps. The call returns MPI\_T\_ERR\_INVALID\_INDEX if the index does not identify a 21valid source. The call returns MPI\_T\_ERR\_NOT\_SUPPORTED if the source does not support 22the ad-hoc generation of timestamps.

#### $^{24}$ Callback Safety Requirements 25

The actions a user is allowed to perform inside a callback function may vary with its 26execution context. As the user has no control over the execution context of specific callback 27function invocations, MPI provides a way to communicate this information using callback 28safety levels. 29

0

30

23

3132

33

34 35

36 37

38 39

40

 $^{41}$ 

Safety Requirement
MPI_T_CB_REQUIRE_NONE
MPI_T_CB_REQUIRE_MPI_RESTRICTED
MPI_T_CB_REQUIRE_THREAD_SAFE
MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE

Table 15.5: Hierarchy of safety requirement levels for event callback routines

Table 15.5 provides the hierarchy of callback safety requirements levels within userdefined callback functions. The MPI implementation provides the safety requirement as an argument to the callback when it is invoked.

42The level of MPI\_T\_CB\_REQUIRE\_NONE is the lowest level and does not impose any 43restrictions on the callback function.

44The level of MPI\_T\_CB\_REQUIRE\_MPI\_RESTRICTED restricts the set of MPI functions 45that can be called from inside the callback to all functions with the prefix MPI\_T as well 46as MPI\_WTICK and MPI\_WTIME.

4748

While some MPI functions are safe to be called inside a callback Advice to users.

1

 $\mathbf{5}$ 

 $\overline{7}$ 8 function used in the MPI tool information interface—which may in some implementations be issued from asynchronous contexts such as signal handlers—this does not imply that those MPI functions are generally safe to be called in asynchronous contexts such as signal handlers. (*End of advice to users.*)

The level of MPI\_T\_CB\_REQUIRE\_THREAD\_SAFE includes all the limitations of MPI\_T\_CB\_REQUIRE\_MPI\_RESTRICTED and additionally requires the callback to be reentrant and thread-safe. This means the callback must allow its execution to be interrupted by or happen concurrently with any other callback including itself.

The level of MPI\_T\_CB\_REQUIRE\_ASYNC\_SIGNAL\_SAFE includes all the limitations of MPI\_T\_CB\_REQUIRE\_THREAD\_SAFE and additionally requires the callback to meet the safety requirements needed to support invocations from asynchronous contexts, such as signal handlers.

Advice to users. It is always safe to assume the highest restrictions for a callback invocation (i.e., MPI\_T\_CB\_REQUIRE\_ASYNC\_SIGNAL\_SAFE). By evaluating the specific requirements at runtime, a tool may obtain more freedom of action within the callback. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will strive to set callback safety requirements to the most permissive level for a given callback invocation. (End of advice to implementors.)

All functions with the prefix MPI\_T, except those listed in Table 15.6, may return the error code MPI\_T\_ERR\_NOT\_ACCESSIBLE to indicate that the user may not access this function at this time. The functions (and their respective PMPI versions) listed in Table 15.6 are exceptions to this rule and shall not return MPI\_T\_ERR\_NOT\_ACCESSIBLE.

MPI_T_EVENT_COPY MPI_T_EVENT_GET_SOURCE	
MPI_T_EVENT_GET_TIMESTAMP	
MPI_T_EVENT_READ MPI_T_PVAR_READ	
MPI_T_PVAR_READRESET	
MPI_T_PVAR_RESET	
MPI_T_PVAR_START	
MPI_T_PVAR_STOP	
MPI_T_PVAR_WRITE MPI_T_SOURCE_GET_TIMESTAMP	

Table 15.6: List of MPI functions that when called from within a callback function may not return MPI\_T\_ERR\_NOT\_ACCESSIBLE

*Rationale.* A call may be implemented in a way that is not safe for all execution contexts of a callback function, e.g., inside a signal handler. An MPI implementation therefore needs a way to communicate its inability to perform a certain action due to the execution context of a callback invocation. (*End of rationale.*)

 $\mathbf{2}$ 

 $\overline{7}$ 

 $^{24}$ 

1 A high-quality implementation shall not return Advice to implementors.  $\mathbf{2}$ MPI\_T\_ERR\_NOT\_ACCESSIBLE except where absolutely necessary. (End of advice to 3 *implementors.*) 4 Advice to users. Users intercepting calls into the MPI tool information interface using 5the PMPI interface must ensure that the safety requirements for the calling context 6 are met. This means that users may have to implement the wrapper with the highest 7 safety level used by the MPI implementation. (End of advice to users.) 8 9 10**Event Type Query Functions** 11An MPI implementation exports a set of N event types through the MPI tool information 12interface. If N is zero, then the MPI implementation does not export any event types; 13 otherwise, the provided event types are indexed from 0 to N-1. This index number is 14 used in subsequent calls to identify a specific event type. 15An MPI implementation is allowed to increase the number of event types during the 16execution of an MPI process. However, MPI implementations are not allowed to change the 17index of an event type or to delete an event type once it has been made visible to the user 18 (e.g., if new event types become available via dynamic loading of additional components in 19the MPI implementation). 20The following function can be used to query the number of event types, *num\_events*: 2122 23MPI\_T\_EVENT\_GET\_NUM(num\_events)  $^{24}$ OUT num\_events returns number of event types (integer) 2526C binding 27int MPI\_T\_event\_get\_num(int \*num\_events) 2829The function MPI\_T\_EVENT\_GET\_INFO provides access to additional information 30 about a specific event type.  $^{31}$ 32 33 3435 36 37 38 39 40 41 4243 444546 4748

MPI\_T\_EVENT\_GET\_INFO(event\_index, name, name\_len, verbosity, array\_of\_datatypes, array\_of\_displacements, num\_elements, enumtype, info, desc, desc\_len, bind)

IN	event_index	index of the event type to be queried between 0 and $num\_events - 1$ (integer)	4 5 6
OUT	name	buffer to return the string containing the name of the event type (string)	7
INOUT	name_len	length of the string and/or buffer for name (integer) $% \left( {{\left[ {{{\left[ {{{\left[ {{\left[ {{\left[ {{\left[ {{{\left[ {{{\left[ {{{\left[ {{\left[ {{{\left[ {{{\left[ {{{\left[ {{{\left[ {{{}}}} \right]}}}} \right.$	9
OUT	verbosity	verbosity level of this event type (integer)	10 11
OUT	array_of_datatypes	array of MPI basic data types used to encode the event data (array of handles)	11 12 13
OUT	array_of_displacements	array of byte displacements of the elements in the event buffer (array of non-negative integers)	14 15
INOUT	num_elements	<pre>length of array_of_datatypes and array_of_displacements arrays (non-negative integer)</pre>	16 17 18
OUT	enumtype	optional descriptor for enumeration information (handle)	19 20
OUT	info	optional info object (handle)	21
OUT	desc	buffer to return the string containing a description of the event type (string)	22 23
ΙΝΟυτ	desc_len	length of the string and/or buffer for desc (integer)	24 25
OUT	bind	type of MPI object to which an event of this type must be bound (integer)	26 27

#### C binding

After a successful call to MPI\_T\_EVENT\_GET\_INFO for a particular event type, subsequent calls to this routine that query information about the same event type must return the same information. If any INOUT or OUT argument to MPI\_T\_EVENT\_GET\_INFO is a NULL pointer, the implementation will ignore the argument and not return a value for the specific argument.

The arguments name and name\_len are used to return the name of the event type as described in Section 15.3.3. If completed successfully, the routine is required to return a name of at least length one. The name of the event type must be unique with respect to all other names for event types used by the MPI implementation.

The argument verbosity returns the verbosity level of the event type (see Section 15.3.1).

The argument array\_of\_datatypes returns an array of MPI datatype handles that describe the elements returned for an instance of the event type with index event\_index. The event data can either be queried element by element with MPI\_T\_EVENT\_READ or copied 48

1 2

28 29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

into a contiguous event buffer with MPI\_T\_EVENT\_COPY. For the latter case, the argument array\_of\_displacements returns an array of byte displacements in the event buffer in ascending order starting with zero.

The user is responsible for the memory allocation for the array\_of\_datatypes and

array\_of\_displacements arrays. The number of elements in each array is supplied by the user
 in num\_elements. If the number of elements used by the event type is larger than the value
 of num\_elements provided by the user, the number of datatype handles and displacements
 returned in the corresponding arrays is truncated to the value of num\_elements passed in
 by the user. If the user passes the NULL pointer for array\_of\_datatypes or

array\_of\_displacements, the respective arguments are ignored. Unless the user passes the
 NULL pointer for num\_elements, the function returns the number of elements required for
 this event type. If the user passes the NULL pointer for num\_elements, the arguments
 num\_elements, array\_of\_datatypes, and array\_of\_displacements are ignored.

<sup>14</sup> MPI can optionally return an enumeration identifier in the enumtype argument, de-<sup>15</sup> scribing the individual elements in the array\_of\_datatypes argument. Otherwise, enumtype <sup>16</sup> is set to MPI\_T\_ENUM\_NULL. If the argument to enumtype provided by the user is the NULL <sup>17</sup> pointer, no enumeration type is returned.

18 MPI can optionally return an info object containing the default hints set for a regis-19tration handle for this event type. If the argument to info provided by the user is the NULL 20pointer, this argument is ignored, otherwise an MPI implementation is required to return 21all hints that are supported by the implementation for a registration handle for this event 22type and have default values specified; any user-supplied hints that were not ignored by the 23implementation; and any additional hints that were set by the implementation. If no such  $^{24}$ hints exist, a handle to a newly created info object is returned that contains no key/value 25pair. The user is responsible for freeing info via MPI\_INFO\_FREE.

The arguments desc and desc\_len are used to return the description of the event type as described in Section 15.3.3. Returning a description is optional. If an MPI implementation does not return a description, the first character for desc must be set to the null character and desc\_len must be set to one at the return from this function.

The parameter bind returns the type of the MPI object to which the event type must be bound or the value MPI\_T\_BIND\_NO\_OBJECT (see Section 15.3.2).

<sup>32</sup> If an event type has an equivalent name across connected MPI processes, the following
 <sup>33</sup> OUT parameters must be identical: verbosity, array\_of\_datatypes, num\_elements, enumtype,
 <sup>34</sup> and bind. The returned description must be equivalent. As the argument

<sup>35</sup> array\_of\_displacements is process dependent, it may differ across connected MPI processes.
 <sup>36</sup> This routine returns MPI\_SUCCESS on success and returns MPI\_T\_ERR\_INVALID\_INDEX
 <sup>37</sup> if event\_index does not match a valid event type index provided by the implementation at
 <sup>38</sup> the time of the call.

39 40

MPI\_T\_EVENT\_GET\_INDEX(name, event\_index)

2	IN	name	name of the event type (string)
8	OUT	event_index	index of the event type (integer)

 $_{46}^{45}$  C binding

```
47 int MPI_T_event_get_index(const char *name, int *event_index)
48
```

MPI\_T\_EVENT\_GET\_INDEX returns the index of an event type identified by a known event type name. The name parameter is provided by the caller, and event\_index is returned by the MPI implementation. The name parameter is a string terminated with a null character.

This routine returns MPI\_SUCCESS on success and returns MPI\_T\_ERR\_INVALID\_NAME if **name** does not match the name of any event type provided by the implementation at the time of the call.

*Rationale.* This routine is provided to enable fast retrieval of an event index by a tool, assuming it knows the name of the event type for which it is looking. The number of event types exposed by the implementation can change over time, so it is not possible for the tool to simply iterate over the list of event types once at initialization. Although using MPI implementation specific event type names is not portable across MPI implementations, tool developers may choose to take this route for lower overhead at runtime because the tool will not have to iterate over the entire set of event types to find a specific one. (*End of rationale.*)

# Handle Allocation and Deallocation

Before the MPI implementation calls a callback function on the occurrence of a specific event, the user needs to register a callback function to be called for that event type and obtain a handle of type MPI\_T\_event\_registration.

# MPI\_T\_EVENT\_HANDLE\_ALLOC(event\_index, obj\_handle, info, event\_registration)

IN	event_index	index of event type for which the registration handle is to be allocated (integer)	
IN	obj_handle	reference to a handle of the MPI object to which this event is supposed to be bound (pointer)	
IN	info	info object (handle)	3
OUT	event_registration	event registration (handle)	:

# C binding

MPI\_T\_EVENT\_HANDLE\_ALLOC creates a *registration handle* for the event type identified by event\_index. Furthermore, if required by the event type, the registration handle is bound to the object referred to by the argument obj\_handle. The argument obj\_handle is ignored if the MPI\_T\_EVENT\_GET\_INFO call for this event type returned MPI\_T\_BIND\_NO\_OBJECT in the argument bind. The user can pass hints for the handle allocation to the MPI implementation via the info argument. The allocated event-registration handle is returned in the argument event\_registration.  $\mathbf{5}$ 

```
1
      MPI_T_EVENT_HANDLE_SET_INFO(event_registration, info)
2
                  event_registration
       INOUT
                                               event registration (handle)
3
       IN
                 info
                                               info object (handle)
4
5
6
      C binding
\overline{7}
      int MPI_T_event_handle_set_info(
8
                     MPI_T_event_registration event_registration, MPI_Info info)
9
          MPI_T_EVENT_HANDLE_SET_INFO updates the hints of the event-registration han-
10
      dle associated with event_registration using the hints provided in info. This operation has
11
      no effect on previously set or defaulted hints that are not specified by info. It also has no
12
      effect on previously set or defaulted hints that are specified by info, but are ignored by the
13
      MPI implementation in this call to MPI_T_EVENT_HANDLE_SET_INFO.
14
15
           Advice to users.
                              Some info items that an implementation can use when it creates
16
           an event-registration handle cannot easily be changed once the registration handle
17
           is created. Thus, an implementation may ignore hints issued in this call that it
18
           would have accepted in a handle allocation call. An implementation may also be
19
           unable to update certain info hints in a call to MPI_T_EVENT_HANDLE_SET_INFO.
20
           MPI_T_EVENT_HANDLE_GET_INFO can be used to determine whether info changes
21
           were ignored by the implementation. (End of advice to users.)
22
23
^{24}
      MPI_T_EVENT_HANDLE_GET_INFO(event_registration, info_used)
25
26
       IN
                  event_registration
                                               event registration (handle)
27
        OUT
                 info_used
                                               info object (handle)
28
29
      C binding
30
      int MPI_T_event_handle_get_info(
^{31}
                     MPI_T_event_registration event_registration,
32
                     MPI_Info *info_used)
33
34
          MPI_T_EVENT_HANDLE_GET_INFO returns a new info object containing the hints of
35
      the event-registration handle associated with event_registration. The current setting of all
36
      hints related to this registration handle is returned in info_used. An MPI implementation
37
      is required to return all hints that are supported by the implementation and have default
38
      values specified; any user-supplied hints that were not ignored by the implementation; and
39
      any additional hints that were set by the implementation. If no such hints exist, a handle
40
      to a newly created info object is returned that contains no key/value pairs. The user is
41
     responsible for freeing info_used via MPI_INFO_FREE.
42
43
44
45
46
47
48
```

MPI_T_EVENT_REGISTER_CALLBACK(event_registration, cb_safety, info, user_data, <sup>1</sup>			
	event_cb_function)		2
INOUT	event_registration	event registration (handle)	3
			4
IN	cb_safety	maximum callback safety level (integer)	5
IN	info	info object (handle)	6
	and date	· · · · · · · · · · · · · · · · · · ·	7
IN	user_data	pointer to a user-controlled buffer	8
IN	event_cb_function	pointer to user-defined callback function (function)	9
	event_eb_runetion	pointer to user defined canoack function (function)	9
			10

# MPL T EVENT REGISTER CALLBACK(event registration cb safety info user data

# C binding

8
<pre>int MPI_T_event_register_callback(</pre>
MPI_T_event_registration event_registration,
<pre>MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data,</pre>
MPI_T_event_cb_function event_cb_function)

MPI\_T\_EVENT\_REGISTER\_CALLBACK associates a user-defined function pointed to by event\_cb\_function with an allocated event-registration handle. The maximum callback safety level supported by the callback function is passed in the argument cb\_safety. The safety levels are defined in Table 15.5. A user can register multiple callback functions for a given event-registration handle, potentially specifying one for each callback safety level. Registering a callback function for a specific callback safety level overwrites any previouslyregistered callback function pointer and info object associated with the event registration for the specific callback safety level. If event\_cb\_function is the NULL pointer, an existing association of a callback function for that callback safety level is removed.

When an event is triggered, the implementation will select from all registered callbacks the callback with the lowest safety level valid in the context in which the callback is invoked. In situations where the required callback safety level exceeds the highest level for which a callback function is registered for a given registration handle, the event instance is dropped.

At callback invocation time, the implementation passes the pointer to a user-defined memory region specified during callback registration with the argument user\_data.

The user can pass hints for the registration of the specified callback function to the MPI implementation via the info argument.

Advice to users. As event instances can be raised as soon as the registration handle is associated with the first callback function, the callback function with the highest callback safety guarantees should be registered before any further registrations for lower callback safety guarantees, to avoid dropped events due to insufficient callback safety guarantees. (End of advice to users.)

The callback function passed to MPI\_T\_EVENT\_REGISTER\_CALLBACK in the argument event\_cb\_function needs to have the following type: typedef void MPI\_T\_event\_cb\_function(MPI\_T\_event\_instance event\_instance, MPI\_T\_event\_registration event\_registration, MPI\_T\_cb\_safety cb\_safety, void \*user\_data);

The argument event\_instance corresponds to a handle for the opaque event-instance 46object of type MPI\_T\_event\_instance. This handle is only valid inside the corresponding 47invocation of the function to which it is passed. The argument event\_registration corresponds 48

11

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

31

32

33 34

35

36

37

38

39 40

41

42

43

44

```
1
      to the event-registration handle returned by MPI_T_EVENT_HANDLE_ALLOC for the user
\mathbf{2}
      function to the same event type and bound object combination. The handle can be used to
3
      identify the specific event registration information, such as event type and bound object, or
4
      even to deallocate the handle from within the callback invocation. The argument cb_safety
\mathbf{5}
      describes the safety requirements the callback function must fulfill in the current invocation.
6
      The argument user_data is the pointer to user-allocated memory that was passed to the MPI
\overline{7}
      implementation during callback registration.
8
9
      MPI_T_EVENT_CALLBACK_SET_INFO(event_registration, cb_safety, info)
10
11
        INOUT
                  event_registration
                                                event registration (handle)
12
        IN
                  cb_safety
                                                callback safety level (integer)
13
        IN
                  info
                                                info object (handle)
14
15
16
      C binding
17
      int MPI_T_event_callback_set_info(
18
                      MPI_T_event_registration event_registration,
19
                      MPI_T_cb_safety cb_safety, MPI_Info info)
20
          MPI_T_EVENT_CALLBACK_SET_INFO updates the hints of the callback function reg-
21
      istered for the callback safety level specified by cb_safety of the event-registration handle
22
      associated with event_registration using the hints provided in info. This operation has no
23
      effect on previously set or defaulted hints that are not specified by info. It also has no effect
24
      on previously set or defaulted hints that are specified by info, but are ignored by the MPI
25
      implementation in this call to MPI_T_EVENT_CALLBACK_SET_INFO.
26
27
28
      MPI_T_EVENT_CALLBACK_GET_INFO(event_registration, cb_safety, info_used)
29
        IN
                  event_registration
                                                event registration (handle)
30
        IN
                  cb_safety
                                                callback safety level (integer)
^{31}
32
        OUT
                  info_used
                                               info object (handle)
33
34
      C binding
35
      int MPI_T_event_callback_get_info(
36
                      MPI_T_event_registration event_registration,
37
                      MPI_T_cb_safety cb_safety, MPI_Info *info_used)
38
          MPI_T_EVENT_CALLBACK_GET_INFO returns a new info object containing the hints
39
      of the callback function registered for the callback safety level specified by cb_safety of the
40
      event-registration handle associated with event_registration. The current set of all hints
41
      related to this callback safety level of the event-registration handle is returned in info_used.
42
      An MPI implementation is required to return all hints that are supported by the imple-
43
      mentation and have default values specified, any user-supplied hints that were not ignored
44
      by the implementation, and any additional hints that were set by the implementation. If
45
      no such hints exist, a handle to a newly created info object is returned that contains no
46
47
      key/value pairs. The user is responsible for freeing info_used via MPI_INFO_FREE.
48
```

To stop the MPI implementation from raising events for a specific registration, a user needs to free the corresponding event-registration handle.

MPI_T_EVENT_HANDLE_FREE(event_registration, user_data, free_cb_function)			
INOUT	event_registration	event registration (handle)	
IN	user_data	pointer to a user-controlled buffer	
IN	free_cb_function	pointer to user-defined callback function (function)	

# C binding

MPI\_T\_EVENT\_HANDLE\_FREE returns MPI\_SUCCESS when deallocation of the handle was initiated successfully and returns MPI\_T\_ERR\_INVALID\_HANDLE if event\_registration does not match a valid allocated event-registration handle at the time of the call. The callback function free\_cb\_function is called by the MPI implementation, when it is able to guarantee that no further event instances for the corresponding eventregistration handle will be raised. If the pointer to free\_cb\_function is the NULL pointer, no user function is invoked after successful deallocation of the event registration handle. The pointer to user-controlled memory provided in the user\_data argument will be passed to the function provided in the free\_cb\_function.

Advice to users. A free-callback function associated with a registration handle should always be prepared to postpone any pending actions, should the provided callback safety requirements exceed those required by the pending actions. (*End of advice to users.*)

#### Handling Dropped Events

Events may occur at times when the MPI implementation cannot invoke the user function corresponding to a matching event handle. An implementation is allowed to buffer such events and delay the callback invocation. If an event occurs at times when the corresponding callback function cannot be called and the corresponding data cannot be buffered, or no callback function meeting the required callback safety level is registered, the event data may be dropped. To discover such data loss, the user can set a handler function for a specific event-registration handle.  $\mathbf{2}$ 

1 2			NDLER(event_registration, dropped_cb_function)
3	INOUT	event_registration	valid event registration (handle)
4	IN	dropped_cb_function	pointer to user-defined callback function (function)
5			
6	C binding	-	
7	int MPI_1	[_event_set_dropped_hand	
8 9		•	ration event_registration, d_cb_function dropped_cb_function)
10			••
11			_HANDLER registers the function the MPI implementation when event information is
12	• •		pecified in event_registration. Subsequent calls to
13			NDLER with the same registration handle will replace
14 15			ns for that registration handle. If the pointer to
16			ter, no data loss is recorded or reported until a new
17	valid callb	ack function is registered.	
18	Advi	ce to users. The invocation	on of the dropped handler callback function may not
19			ime the event was actually lost. (End of advice to
20	usera	-	
21 22	נרדו		
22		dropped_cb_function passed to	MPI_T_EVENT_SET_DROPPED_HANDLER in the to have the following type:
24			d_cb_function(MPI_Count count,
25	-JF		ration event_registration, int source_index,
26		MPI_T_cb_safety cb	_safety, void *user_data);
27	The a	rgument event_registration (	corresponds to the event registration handle to which
28 29			argument count provides a best effort estimation of
30		_	red event callback corresponding to <code>event_registration</code>
31			istration of the dropped-callback handler or the last
32			llback handler. The source_index provides the index ponding event information. The argument cb_safety
33			callback function must fulfill in the current invocation.
34 35			described in Table 15.5. The argument user_data is
36	the pointe	r to user-allocated memory	that was passed to the MPI implementation during
37		-	ack is registered for safety requirement levels that an
38	-		opped handler callback function for a specific event,
39	the corresp	ponding dropped handler ca	ullback function will not be invoked.
40	Advi	ce to users. A callback fu	nction for dropped events associated with a registra-
41 42			repared to postpone any pending actions, should the
42 43	-		ments exceed those required by the pending actions.
44	(End	l of advice to users.)	
45	Advi	ce to implementors. A hig!	h-quality implementation should strive to find a good
46			tion, completeness of information, and the freedom of
47		0	the callback function for dropped events associated
48	with	a registration handle. (End	d of advice to implementors.)

If dropped event notifications have been observed for a specific source since the last event notification of that source, the corresponding dropped handler callback function must be called before other events are raised for that source. This means in a sequence of five events E1 to E5 from the same source, where E3 and E4 were dropped, any handler function set through MPI\_T\_EVENT\_SET\_DROPPED\_HANDLER for event-registration handles associated with E3 or E4 must be called before E5 is raised.

### Reading Event Data

In event callbacks, the parameter event\_instance provides access to the per-instance event data, i.e., the data encoded by the specific event type for this instance. The user can obtain event data as well as event meta data, such as a time stamp and the source, by providing this handle to the respective query functions. The event-instance handle is invalid beyond the scope of the current invocation of the callback function to which it is provided.

The callback function argument event\_registration identifies the registration handle that was used to register the callback function.

The callback function argument cb\_safety indicates the requirements for the specific callback invocation. The value is one of the safety requirements levels described in Table 15.5. The argument user\_data passes the pointer provided by the user during callback registration back to the function call.

Advice to users. Depending on the registered event and usage of MPI by the application, a callback function may be invoked with high frequency. Users should therefore strive to minimize the amount of work done inside callback functions. Furthermore, the time spent in a callback function may influence the capability of an implementation to buffer events; long execution times may lead to an increased number of dropped events. (End of advice to users.)

MPI provides the following function calls to access data of a specific event instance and its corresponding meta data (such as its time and source).

#### MPI\_T\_EVENT\_READ(event\_instance, element\_index, buffer)

		, , , , , , , , , , , , , , , , , , ,	33
IN	event_instance	event-instance handle provided to the callback	34
		function (handle)	35
IN	element_index	index into the array of datatypes of the item to be	36
		queried (integer)	37
OUT	buffer	pointer to a memory location to store the item data	38
		(choice)	39
			40

# C binding

MPI\_T\_EVENT\_READ allows users to copy one element of the event data to a userspecified buffer at a time.

The event\_instance argument identifies the event instance to query. It is erroneous 47 to provide any other event-instance handle to the call than the one passed by the MPI 48

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6 7 8

9

10

11

12

13

14

15

16

17

18

19

20 21

22

23

 $^{24}$ 

25

26

27 28

29

30 31 32

41

42

43

44

45

must poi		nction in which the data is read. The <b>buffer</b> argument ne MPI implementation can copy the element of the event x.			
MPI_T_E	EVENT_COPY(event_insta	nce, buffer)			
IN	event_instance	event instance provided to the callback function (handle)			
OUT	buffer	user-allocated buffer for event data (choice)			
C bindi	•	vent_instance event_instance, void *buffer)			
The user type, wh the corre between location	r must assure that the b ich can be computed fro sponding call to MPI_T_E individual elements of th	the event data as a whole into the user-provided buffer. uffer is of at least the size of the extent of the event on the type and displacement information returned by EVENT_GET_INFO. The data may include padding bytes e event data in the buffer. A user can reconstruct the ained in the buffer through the information returned by			
Advice to implementors. An implementation should strive to use an appropriately compact representation when copying event instance data to a user buffer via MPI_T_EVENT_COPY to reduce the amount of memory required for the user buffer. (End of advice to implementors.)					
Reading I	Event Meta Data				
	al to the specific event dat across all event types car	ta encoded by each event type, supplemental information be queried.			
MPI_T_E	EVENT_GET_TIMESTAM	P(event_instance, event_timestamp)			
IN	event_instance	event instance provided to the callback function (handle)			
OUT	event_timestamp	timestamp the event was observed (integer)			
C bindi int MPI	•	(MPI_T_event_instance event_instance, t_timestamp)			
tially obs instance	served by the implements to query. It is erroneous	TAMP returns the timestamp of when the event was ini- ation. The event_instance argument identifies the event is to provide any other handle to the call than the one to the callback function in which the timestamp is read.			
	-	plementation may postpone the call to the user's callback call to MPI_T_EVENT_GET_TIMESTAMP may yield a			

timestamp in the past that is closer to the time the event was initially observed, as opposed to a timestamp captured during callback function invocation. (*End of advice to users.*)

Advice to implementors. A high-quality implementation will return a timestamp as close as possible to the earliest time the event was observed by the MPI implementation. (*End of advice to implementors.*)

An event may be raised from different components acting as event sources in the MPI implementation. A source in this context is an abstract concept that helps to define partial ordering of raised events, as each source provides its own ordering guarantees. A source describes the entity that raises the event, rather than the origin of the data.

To identify the source of an event instance, the user can query the index of the source within the corresponding event callback function invocation.

Advice to implementors. An excessive number of event sources may negatively impact performance of a tool due to per-source overhead in event handling. (End of advice to implementors.)

# MPI\_T\_EVENT\_GET\_SOURCE(event\_instance, source\_index) IN event\_instance event\_instance event instance provided to the callback function (handle) OUT source\_index index identifying the source (integer)

#### C binding

The event\_instance argument identifies the event instance to query. It is erroneous to provide any other event-instance handle to the call than the one passed by the MPI implementation to the callback function in which the source is queried.

The source\_index argument returns the index of the source of the event instance. It can be used to query more information on the source using MPI\_T\_SOURCE\_GET\_INFO.

*Rationale.* Event callback function invocations are associated with a source to enable chronological processing of events on the tool side, when required, while retaining low overhead on the side of the MPI implementation. (*End of rationale.*)

# 15.3.9 Variable Categorization

MPI implementations can optionally group performance and control variables into categories to express logical relationships between various variables. For example, an MPI implementation could group all control and performance variables that refer to message transfers in the MPI implementation and thereby distinguish them from variables that refer to local resources such as memory allocations or other interactions with the operating system.

Categories can also contain other categories to form a hierarchical grouping. Categories <sup>47</sup> can never include themselves, either directly or transitively within other included categories. <sup>48</sup>

 $^{24}$ 

 $41 \\ 42$ 

 $45 \\ 46$ 

Expanding on the example above, this allows MPI to refine the grouping of variables referring
 to message transfers into variables to control and to monitor message queues, message
 matching activities and communication protocols. Each of these groups of variables would
 be represented by a separate category and these categories would then be listed in a single
 category representing variables for message transfers.

<sup>6</sup> The category information may be queried in a fashion similar to the mechanism for <sup>7</sup> querying variable information. The MPI implementation exports a set of N categories via <sup>8</sup> the MPI tool information interface. If N = 0, then the MPI implementation does not export <sup>9</sup> any categories, otherwise the provided categories are indexed from 0 to N - 1. This index <sup>10</sup> number is used in subsequent calls to functions of the MPI tool information interface to <sup>11</sup> identify the individual categories.

<sup>12</sup> An MPI implementation is permitted to increase the number of categories during the <sup>13</sup> execution of an MPI program when new categories become available through dynamic load-<sup>14</sup> ing. However, MPI implementations are not allowed to change the index of a category or <sup>15</sup> delete it once it has been added to the set.

<sup>16</sup> Similarly, MPI implementations are allowed to add variables to categories, but they
 <sup>17</sup> are not allowed to remove variables from categories or change the order in which they are
 <sup>18</sup> returned.

<sup>20</sup> Category Query Functions

The following function can be used to query the number of categories, num\_cat.

```
MPI_T_CATEGORY_GET_NUM(num_cat)
```

OUT num\_cat current number of categories (integer)

# <sup>28</sup> C binding

19

21

23 24

25

26 27

30

```
<sup>29</sup> int MPI_T_category_get_num(int *num_cat)
```

Individual category information can then be queried by calling the following function:

MPI_T_C	ATEGORY_GET_INFC num_pvars, num	D(cat_index, name, name_len, desc, desc_len, num_cvars, n_categories)	1 2
IN	cat_index	index of the category to be queried (integer)	3
OUT	name	buffer to return the string containing the name of the category (string)	4 5 6
INOUT	name_len	length of the string and/or buffer for name (integer)	7
OUT	desc	buffer to return the string containing the description of the category (string)	8 9 10
INOUT	desc_len	length of the string and/or buffer for $desc$ (integer)	11
OUT	num_cvars	number of control variables in the category (integer)	12
OUT	num_pvars	number of performance variables in the category (integer)	13 14 15
OUT	num_categories	number of categories contained in the category (integer)	16 17 18
described The n unique wi	in Section 15.3.3. routine is required to th respect to all other	name_len are used to return the name of the category as return a name of at least length one. This name must be names for categories used by the MPI implementation.	
unique wi If any	th respect to all other $y$ OUT parameter to N	9	25 26 27 28 29
described	in Section $15.3.3$ .	$\ensuremath{esc\_len}$ are used to return the description of the category as	30 31
descriptio		optional. If an MPI implementation decides not to return a for desc must be set to the null character and desc_len must his call.	32 33 34
categories		umber of control variables, performance variables and other ried category in the arguments num_cvars, num_pvars, and	35 36
If the		is equivalent across connected MPI processes, then the re-ivalent.	37 38 39
		I EVENTS(cot index num quanta)	40 41
		1_EVENTS(cat_index, num_events)	42
IN	cat_index	index of the category to be queried (integer)	43
OUT	num_events	number of event types in the category (integer)	44 45
C bindin	ıg		46
	•	_events(int cat_index, int *num_events)	47
			48

MPI_T_C	ATEGORY_GET_IND	EX(name, cat_index)
IN	name	the name of the category (string)
OUT	cat_index	the index of the category (integer)
C bindinint MPI_	-	<pre>lex(const char *name, int *cat_index)</pre>
given a ki is returne null chara This	nown category name. ed by the MPI implem acter. routine returns MPI_S oes not match the na	_INDEX is a function for retrieving the index of a cate The name parameter is provided by the caller, and cat_i entation. The name parameter is a string terminated with SUCCESS on success and returns MPI_T_ERR_INVALID_N time of any category provided by the implementation at
$\operatorname{pos}$	sible for the tool to si	osed by the implementation can change over time, so it is mply iterate over the list of categories once at initializa
pos Alt MP at 1	sible for the tool to sin hough using MPI imp l implementations, too	mply iterate over the list of categories once at initializate elementation specific category names is not portable at ol developers may choose to take this route for lower over ool will not have to iterate over the entire set of category
pos Alt MP at 1 to f	sible for the tool to simulations in the tool to simulate the simulation of the simu	mply iterate over the list of categories once at initializate elementation specific category names is not portable at ol developers may choose to take this route for lower over ool will not have to iterate over the entire set of catego and of rationale.)
pos Alt MP at 1 to f	sible for the tool to sight hough using MPI implementations, too runtime because the t ind a specific one. (En Member Query Functio	mply iterate over the list of categories once at initializate elementation specific category names is not portable at ol developers may choose to take this route for lower over ool will not have to iterate over the entire set of catego and of rationale.)
pos Alt MP at 1 to f	sible for the tool to sight hough using MPI implementations, too runtime because the t ind a specific one. (En Member Query Functio	mply iterate over the list of categories once at initializate elementation specific category names is not portable ac ol developers may choose to take this route for lower over lool will not have to iterate over the entire set of catego and of rationale.)
pos Alt MP at 1 to f Category MPI_T_C	sible for the tool to sighough using MPI implementations, too runtime because the t and a specific one. (Each Member Query Function CATEGORY_GET_CVA	mply iterate over the list of categories once at initializate elementation specific category names is not portable ac ol developers may choose to take this route for lower overl ool will not have to iterate over the entire set of catego <i>nd of rationale.</i> ) ons ARS(cat_index, len, indices) index of the category to be queried, in the range
pos Alt MP at 1 to f Category MPI_T_C IN	sible for the tool to sighough using MPI implementations, too nuntime because the t a specific one. (Eacher Member Query Function ATEGORY_GET_CVA cat_index	<pre>mply iterate over the list of categories once at initialization and the specific category names is not portable and of developers may choose to take this route for lower overloool will not have to iterate over the entire set of categories and of rationale.)  MRS(cat_index, len, indices)  index of the category to be queried, in the range from 0 to num_cat - 1 (integer)</pre>
pos Alt MP at r to f Category MPI_T_C IN IN OUT	sible for the tool to sighough using MPI implementations, too nuntime because the t and a specific one. (Each Member Query Function CATEGORY_GET_CVA cat_index len indices	<pre>mply iterate over the list of categories once at initializa elementation specific category names is not portable ac elementation specific category to be end elementation of elementation acceleration and elementation of the indices array (integer)</pre>

MPI_T_C	ATEGORY_GET_PVARS(c	at_index, len, indices)	1
IN	cat_index	index of the category to be queried, in the range from 0 to $num_cat - 1$ (integer)	2 3
IN	len	the length of the indices array (integer)	4 5
OUT	indices	an integer array of size len, indicating performance variable indices (array of integers)	6 7
			8

# MPL T CATEGORY GET PVARS(cat index lon indicas)

# C binding

<pre>int MPI_T_category_get_pvars(int cat_index, int len, int indices[]</pre>	int	MPI_T	_category_	_get_pvars	s(int	cat_	index,	int	len,	int	indices	[])
---	-----	-------	------------	------------	-------	------	--------	-----	------	-----	---------	-----

MPI\_T\_CATEGORY\_GET\_PVARS can be used to query which performance variables are contained in a particular category. A category contains zero or more performance variables.

# MPI\_T\_CATEGORY\_GET\_EVENTS(cat\_index, len, indices)

IN	cat_index	index of the category to be queried, in the range	18
		from 0 to $num\_cat - 1$ (integer)	19
IN	len	the length of the indices array (integer)	20
OUT	indices	an interest among of size last indication quant turns	21
001	mulces	an integer array of size len, indicating event type	22
		indices (array of integers)	23

# C binding

int MPI_T_category_ge	t_events(int	cat_index,	int len,	int	indices[])
-----------------------	--------------	------------	----------	-----	------------

MPI\_T\_CATEGORY\_GET\_EVENTS can be used to query which event types are contained in a particular category. A category contains zero or more event types.

# MPI\_T\_CATEGORY\_GET\_CATEGORIES(cat\_index, len, indices)

IN	cat_index	index of the category to be queried, in the range from 0 to $num_cat - 1$ (integer)	:
IN	len	the length of the indices array (integer)	:
OUT	indices	an integer array of size $len,$ indicating category	:
		indices (array of integers)	:

## C binding

int MPI\_T\_category\_get\_categories(int cat\_index, int len, int indices[])

MPI\_T\_CATEGORY\_GET\_CATEGORIES can be used to query which other categories are contained in a particular category. A category contains zero or more other categories.

As mentioned above, MPI implementations can grow the number of categories as well as the number of variables or other categories within a category. In order to allow users of the MPI tool information interface to check quickly whether new categories have been added or new variables or categories have been added to a category, MPI maintains an

 $^{24}$ 

update number that is monotonically increasing during the execution and is returned by  $\mathbf{2}$ the following function: 4 MPI\_T\_CATEGORY\_CHANGED(update\_number) 6 OUT update\_number update number (integer) 8 C binding int MPI\_T\_category\_changed(int \*update\_number) 10 If two calls to this routine return the same update number, it is guaranteed that the 11 category information has not changed between the two calls. If the update number retrieved 12from the second call is higher, then some categories have been added or expanded. 13 The index values returned in indices by MPI\_T\_CATEGORY\_GET\_CVARS, 14MPI\_T\_CATEGORY\_GET\_PVARS, MPI\_T\_CATEGORY\_GET\_EVENTS, and 15MPI\_T\_CATEGORY\_GET\_CATEGORIES can be used as input to 16MPI\_T\_CVAR\_GET\_INFO, MPI\_T\_PVAR\_GET\_INFO, MPI\_T\_EVENT\_GET\_INFO, and 17MPI\_T\_CATEGORY\_GET\_INFO, respectively. 18 The user is responsible for allocating the arrays passed into the functions 19MPI\_T\_CATEGORY\_GET\_CVARS, MPI\_T\_CATEGORY\_GET\_PVARS, 20MPI\_T\_CATEGORY\_GET\_EVENTS, and MPI\_T\_CATEGORY\_GET\_CATEGORIES. Start-21ing from array index 0, each function writes up to len elements into the array. If the 22category contains more than len elements, the function returns an arbitrary subset of size 23len. Otherwise, the entire set of elements is returned in the beginning entries of the array,  $^{24}$ and any remaining array entries are not modified. 2526Return Codes for the MPI Tool Information Interface 2715.3.10 28All functions defined as part of the MPI tool information interface return an integer error 29code (see Table 15.7) to indicate whether the function was completed successfully or was 30 aborted. In the latter case, the error code indicates the reason for not completing the  $^{31}$ routine. Such errors neither impact the execution of the MPI process nor invoke MPI error 32 handlers. The MPI process continues executing regardless of the return code from the 33 call. The MPI implementation is not required to check all user-provided parameters; if a 34 user passes invalid parameter values to any routine the behavior of the implementation is 35 undefined. 36 All error codes with the prefix MPI\_T\_ must be unique values and cannot overlap with 37 any other error codes or error classes returned by the MPI implementation. Further, they 38 shall be treated as MPI error classes as defined in Section 9.4 and follow the same rules and 39 restrictions. In particular, they must satisfy: 40

41 42

43

45

 $0 = MPI_SUCCESS < MPI_T_ERR_XXX < MPI_ERR_LASTCODE.$ 

#### 44Profiling Interface 15.3.11

All requirements for the profiling interfaces, as described in Section 15.2, also apply to 46 the MPI tool information interface. All rules, guidelines, and recommendations from Sec-47tion 15.2 apply equally to calls defined as part of the MPI tool information interface. 48

1

3

5

7

Return Code	Description
Return Codes for All Functions in t	he MPI Tool Information Interface
MPI_SUCCESS	Call completed successfully
MPI_T_ERR_INVALID	Invalid or bad parameter value(s)
MPI_T_ERR_MEMORY	Out of memory
MPI_T_ERR_NOT_INITIALIZED	Interface not initialized
MPI_T_ERR_CANNOT_INIT	Interface not in the state to be initialized
MPI_T_ERR_NOT_ACCESSIBLE	Requested functionality not accessible
Return Codes for Datatype Function	ns: MPI_T_ENUM_*
MPI_T_ERR_INVALID_INDEX	The enumeration index is invalid
Return Codes for Variable, Categor	y, and Event Query Functions: MPI_T_*_GET_*
MPI_T_ERR_INVALID_INDEX	The variable or category index is invalid
MPI_T_ERR_INVALID_NAME	The variable or category name is invalid
Return Codes for Handle Functions	
MPI_T_ERR_INVALID_INDEX	The variable index is invalid
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_OUT_OF_HANDLES	No more handles available
Return Codes for Performance Experim	nent Session Functions: MPI_T_PVAR_SESSION_*
MPI_T_ERR_OUT_OF_SESSIONS	No more sessions available
MPI_T_ERR_INVALID_SESSION	Session argument is not a valid session
Return Codes for Control Variable .	Access Functions: MPI_T_CVAR_{READ WRITE}
MPI_T_ERR_CVAR_SET_NOT_NOW	Variable cannot be set at this moment
MPI_T_ERR_CVAR_SET_NEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
Return Codes for Performance Varia	able Access and Control:
MPI_T_PVAR_{START STOP READ	D WRITE RESET READREST}
MPI_T_ERR_INVALID_HANDLE	The handle is invalid
MPI_T_ERR_INVALID_SESSION	Performance experiment session argument is not valid
MPI_T_ERR_PVAR_NO_STARTSTOP	Variable cannot be started or stopped (for
	MPI_T_PVAR_START and MPI_T_PVAR_STOP)
MPI_T_ERR_PVAR_NO_WRITE	Variable cannot be written or reset (for
	MPI_T_PVAR_WRITE and MPI_T_PVAR_RESET)
MPI_T_ERR_PVAR_NO_ATOMIC	Variable cannot be read and written atomically (for
	MPI_T_PVAR_READRESET)
Return Codes for Source Functions:	
MPI_T_ERR_INVALID_INDEX	The source index is invalid
MPI_T_ERR_NOT_SUPPORTED	Requested functionality not supported
Return Codes for Category Function	
MPI_T_ERR_INVALID_INDEX	The category index is invalid
	The caregory much is mitallu

Table 15.7: Return codes used in functions of the MPI tool information interface

# Chapter 16

# **Deprecated Interfaces**

# 16.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI\_COMM\_CREATE\_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 19.3.7. The language bindings are modified.

MPI\_KEYVAL\_CREATE(copy\_fn, delete\_fn, keyval, extra\_state)

IN	copy_fn	Copy callback function for keyval
IN	delete_fn	Delete callback function for keyval
OUT	keyval	key value for future access (integer)
IN	extra_state	Extra state for callback functions

### C binding

For this routine, an interface within the mpi\_f08 module was never defined.

# Fortran binding MPI\_KEYVAL\_CREATE(COPY\_FN, DELETE\_FN, KEYVAL, EXTRA\_STATE, IERROR) EXTERNAL COPY\_FN, DELETE\_FN INTEGER KEYVAL, EXTRA\_STATE, IERROR

The copy\_fn function is invoked when a communicator is duplicated by 40 MPI\_COMM\_DUP. copy\_fn should be of type MPI\_Copy\_function, which is defined as follows: typedef int MPI\_Copy\_function(MPI\_Comm oldcomm, int keyval, void \*extra\_state, void \*attribute\_val\_in, 

void \*attribute\_val\_out, int \*flag);

A Fortran declaration for such a function is as follows: For this routine, an interface within the mpi\_f08 module was never defined.

1	SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
2	ATTRIBUTE_VAL_OUT, FLAG, IERR)
3	INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
4	ATTRIBUTE_VAL_OUT, IERR
5	LOGICAL FLAG
6	
7	copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or
8	Fortran; MPI_NULL_COPY_FN is a function that does nothing other than return $flag = 0$
9	and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets $flag = 1$ , re-
10	turns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note that
11	MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.
12	Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn
13	function is invoked when a communicator is deleted by $MPI\_COMM\_FREE$ or when a call is
14	made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function,
15	which is defined as follows:
16	<pre>typedef int MPI_Delete_function(MPI_Comm comm, int keyval,</pre>
17	<pre>void *attribute_val, void *extra_state);</pre>
18	A Fortran declaration for such a function is as follows:
19	For this routine, an interface within the mpi_f08 module was never defined.
20	For this fourne, an interface within the mp1_100 module was never defined.
21	SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
22	INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
23	$delete_fn$ may be specified as MPI_NULL_DELETE_FN from either C or Fortran;
24	MPI_NULL_DELETE_FN is a function that does nothing other than return MPI_SUCCESS.
25	Note that MPI_NULL_DELETE_FN is also deprecated.
26	Note that WHI_NOEL_DELETE_IN is also deprecated.
27	The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL
28	in MPI-2.0. The language independent definition of the deprecated function is the same as
29	the new function, except for the function name. The language bindings are modified.
30	
31	
32	MPI_KEYVAL_FREE(keyval)
33	INOUT keyval Frees the integer key value (integer)
34	
35	Chinding
36	C binding int MPI_Keyval_free(int *keyval)
37	Int MPI_KeyVal_Ifee(Int *keyVal)
38	For this routine, an interface within the mpi_f08 module was never defined.
39	Fortran binding
40	MPI_KEYVAL_FREE(KEYVAL, IERROR)
41	
42	INTEGER KEYVAL, IERROR
43	
44	The following function is deprecated and is superseded by $MPI\_COMM\_SET\_ATTR$ in
45	MPI-2.0. The language independent definition of the deprecated function is the same as the
46	new function, except for the function name. The language bindings are modified.
47	
48	

MPI_ATTI	R_PUT(comm, keyval, attribute	e_val)	1
INOUT	comm	communicator to which attribute will be attached (handle)	2 3 4
IN	keyval	key value, as returned by MPI_KEYVAL_CREATE (integer)	4 5 6
IN	attribute_val	attribute value	7
			8 9
C binding int MPI_A	•	nt keyval, void *attribute_val)	10 11
For this re	outine, an interface within the	mpi_f08 module was never defined.	12
Fortran b	oinding		13
	PUT(COMM, KEYVAL, ATTRIB	-	14 15
INTEC	ER COMM, KEYVAL, ATTRIBU	TE_VAL, IERROR	16
			17
		d and is superseded by MPI_COMM_GET_ATTR in	18
		nition of the deprecated function is the same as the ame. The language bindings are modified.	19
new runeu	on, except for the function ne	ane. The language bindings are mounted.	20 21
			22
	R_GET(comm, keyval, attribute	_,	23
IN	comm	communicator to which attribute is attached (handle)	24
IN	keyval	key value (integer)	25
OUT	attribute_val	attribute value, unless $flag = false$	26 27
OUT	flag	true if an attribute value was extracted; $false$ if no	28
		attribute is associated with the key	29
C bindin	a.		30 31
	•	nt keyval, void *attribute_val, int *flag)	32
	-	<pre>mpi_f08 module was never defined.</pre>	33
		mpi_100 module was never defined.	34
Fortran k	0		35
	_GET(COMM, KEYVAL, ATTRIB GER COMM, KEYVAL, ATTRIBU		$\frac{36}{37}$
	CAL FLAG	IE_VAL, IEMION	38
			39
The fo	ollowing function is deprecated	and is superseded by MPI_COMM_DELETE_ATTR	40
		definition of the deprecated function is the same as	41 42
the new function, except for the function name. The language bindings are modified.			
			44
			45
			46
			47

```
1
     MPI_ATTR_DELETE(comm, keyval)
2
       INOUT
                 comm
                                               communicator to which attribute is attached (handle)
3
       IN
                 keyval
                                              The key value of the deleted attribute (integer)
4
5
6
     C binding
7
      int MPI_Attr_delete(MPI_Comm comm, int keyval)
8
     For this routine, an interface within the mpi_f08 module was never defined.
9
10
     Fortran binding
11
     MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
          INTEGER COMM, KEYVAL, IERROR
12
13
14
      16.2
             Deprecated since MPI-2.2
15
16
      The entire set of C++ language bindings was deprecated as of MPI-2.2 and removed in
17
     MPI-3.0. See Chapter 17, Removed Interfaces for more information.
18
19
          The following function typedefs have been deprecated and are superseded by new
20
      names. Other than the typedef names, the function signatures are exactly the same; the
21
      names were updated to match conventions of other function typedef names.
22
23
                     Deprecated Name
                                                New Name
^{24}
                                                MPI_Comm_errhandler_function
                     MPI_Comm_errhandler_fn
25
                     MPI_File_errhandler_fn
                                                MPI_File_errhandler_function
26
                     MPI_Win_errhandler_fn
                                                MPI_Win_errhandler_function
27
28
      16.3
             Deprecated since MPI-4.0
29
30
      Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed
^{31}
     in a future version of the MPI specification.
32
33
          The following function is deprecated and is superseded by the new
34
      MPI_INFO_GET_STRING call in MPI-4.0.
35
36
37
      MPI_INFO_GET(info, key, valuelen, value, flag)
38
       IN
                 info
                                              info object (handle)
39
       IN
                                              kev (string)
                  kev
40
41
                 valuelen
       IN
                                              length of value associated with key (integer)
42
       OUT
                 value
                                              value (string)
43
       OUT
                 flag
                                              true if key defined, false if not (logical)
44
45
46
      C binding
47
      int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,
48
                     int *flag)
```

Fortran	2008 binding		1
MPI_Info	_get(info, key, v	aluelen, value, flag, ierror)	2
	E(MPI_Info), INTEN		3
	RACTER(LEN=*), INT	·	4
	EGER, INTENT(IN) :		5
		n), INTENT(OUT) :: value	6
	CAL, INTENT(OUT)	-	7 8
	GER, UPIIUNAL, IN	TENT(OUT) :: ierror	9
Fortran	binding		10
		ALUELEN, VALUE, FLAG, IERROR)	11
	EGER INFO, VALUELE		12
	RACTER*(*) KEY, VA	LUE	13
LUG	ICAL FLAG		14
This	function retrieves the	ne value associated with key in a previous call to	15
MPI_INF	O_SET. If such a ke	y exists, it sets flag to true and returns the value in value,	16
	-	d leaves value unchanged. valuelen is the number of characters	17
		than the actual size of the value, the value is truncated. In	18
		s than the amount of allocated space to allow for the null	19 20
terminate			20 21
		MAX_INFO_KEY, the call is erroneous. GET is allowed to be called at any time, following the descrip-	22
		t is always available in Section 11.4.1.	23
	m r functionanty tha	t is always available in Section 11.4.1.	24
The	following function is	s deprecated and is superseded by the new	25
MPI_INF	O_GET_STRING call	in MPI-4.0.	26
			27
MPI INF	O GET VALUELEN(i	nfo, key, valuelen, flag)	28
	info		29
IN		info object (handle)	30 31
IN	key	key (string)	32
OUT	valuelen	length of value associated with $key\xspace$ (integer)	33
OUT	flag	true if key defined, false if not (logical)	34
			35
C bindi	ng		36
int MPI_	Info_get_valuelen	(MPI_Info info, const char *key, int *valuelen,	37
	int *flag)		38
Fortran	2008 binding		39
		o, key, valuelen, flag, ierror)	40
	E(MPI_Info), INTEN		41 42
	RACTER(LEN=*), INT		42
INTE	EGER, INTENT(OUT)	:: valuelen	43
	CAL, INTENT(OUT)	0	45
INTE	EGER, OPTIONAL, IN	TENT(OUT) :: ierror	46
Fortran	binding		47
MPI_INFO	_GET_VALUELEN(INF	O, KEY, VALUELEN, FLAG, IERROR)	48

12	INTEGER INFO, VALUELEN CHARACTER*(*) KEY	, IERROR	
3	LOGICAL FLAG		
4 5 6 7 8	Retrieves the length of the value associated with key. If key is defined, valuelen is set to the length of its associated value and flag is set to true. If key is not defined, valuelen is not touched and flag is set to false. The length returned in C does not include the end-of-string character.		
9 10 11	If key is larger than MPI_MAX_INFO_KEY, the call is erroneous. The function MPI_INFO_GET_VALUELEN is allowed to be called at any time, following the description for MPI functionality that is always available in Section 11.4.1.		
12 13 14	The following return code MPI-4.0.	has been dep	precated and is superseded by a new name in
15 16 17	Deprecated MPI_T_ERR_IN		Replacement Name MPI_T_ERR_INVALID_INDEX
18 19 20 21 22	<pre>storage_size() and c_sizeof() int</pre>	rinsic function	leprecated because the Fortran language s provide similar functionality. Note that while h bytes, <b>storage_size()</b> provides the size in bits.
23 24 25	MPI_SIZEOF(x, size)		
25 26	IN x	a F	Fortran variable of numeric intrinsic type (choice)
27 28	OUT size	size	e of machine representation of that type (integer)
29 30 31 32 33	Fortran 2008 binding MPI_Sizeof(x, size, ierror TYPE(*), DIMENSION() INTEGER, INTENT(OUT) : INTEGER, OPTIONAL, INT	:: x : size	ierror
34		ENI(001)	191101
35	Fortran binding		
36	MPI_SIZEOF(X, SIZE, IERROF <type> X</type>	.)	
37	INTEGER SIZE, IERROR		
38			
39 40 41	This function returns the variable. It is a generic Fortrar	-	s of the machine representation of the given has a Fortran binding only.
42 43 44		rray argumen	ar to the C <i>sizeof</i> operator but behaves slightly t, it returns the size of the base element, not <i>vice to users.</i> )
45 46 47 48	Rationale. This function useful. (End of rationale.		ble in other languages because it would not be

# Chapter 17

# **Removed Interfaces**

# 17.1 Removed MPI-1 Bindings

# 17.1.1 Overview

The following MPI-1 bindings were deprecated as of MPI-2 and are removed in MPI-3. They may be provided by an implementation for backwards compatibility, but are not required. Removal of these bindings affects all language-specific definitions thereof. Only the language-neutral bindings are listed when possible.

## 17.1.2 Removed MPI-1 Functions

Table 17.1 shows the removed MPI-1 functions and their replacements.

Table 17.1: Removed MPI	-1 functions and their replacements	
Removed	MPI-2 Replacement	
MPI_ADDRESS	MPI_GET_ADDRESS	
MPI_ERRHANDLER_CREATE	MPI_COMM_CREATE_ERRHANDLER	
MPI_ERRHANDLER_GET	MPI_COMM_GET_ERRHANDLER	
MPI_ERRHANDLER_SET	MPI_COMM_SET_ERRHANDLER	
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT	
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED	
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR	
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT	
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT	
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT	

# 17.1.3 Removed MPI-1 Datatypes

Table 17.2 shows the removed MPI-1 datatypes and their replacements.

# 17.1.4 Removed MPI-1 Constants

Table 17.3 shows the removed MPI-1 constants. There are no replacements.

Table 17.2: Removed MPI-1 datatypes. The indicated routine may be used for changing the lower and upper bound respectively.

4	Removed MPI-2 Replacement
5	MPI_LB MPI_TYPE_CREATE_RESIZED
6	MPI_UB MPI_TYPE_CREATE_RESIZED
7	
8	
9	
10	Table 17.3: Removed MPI-1 constants
11	Removed MPI-1 Constants
12	C type: const int (or unnamed enum)
13	Fortran type: INTEGER
14	MPI_COMBINER_HINDEXED_INTEGER
15	MPI_COMBINER_HVECTOR_INTEGER
16	MPI_COMBINER_STRUCT_INTEGER
17	
18	
19	17.1.5 Demoural MDI 1 Callback Directory as
20 21	17.1.5 Removed MPI-1 Callback Prototypes
21	Table 17.4 shows the removed MPI-1 callback prototypes and their replacements.
22	
24	Table 17.4: Removed MPI-1 callback prototypes and their replacements
25	
26	Removed MPI-2 Replacement
27	MPI_Handler_function MPI_Comm_errhandler_function
28	
29	
30	
31	17.2 C++ Bindings
32	
33	
34	The $C++$ bindings were deprecated as of MPI-2.2. The $C++$ bindings are removed in
35	MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by
36	an implementation as described in the MPI-2.2 standard.
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

1

 $^{2}$ 

## Chapter 18

# Semantic Changes and Warnings

This chapter lists semantic changes that have been introduced into the MPI Standard as well as warnings that could potentially impact program behavior. In addition to those listed here, Chapter 17 also lists changes and backward incompatibilities caused by removing interfaces. Unlike Chapter 17, the changes in this chapter did not go through a deprecation process.

### 18.1 Semantic Changes

This section describes semantics that have changed in a way that would potentially cause an MPI program to behave differently when using this version of the MPI Standard without changing the program's code.

#### 18.1.1 Semantic Changes Starting in MPI-4.0

MPI\_COMM\_DUP and MPI\_COMM\_IDUP no longer propagate info hints from the input communicator to the output communicator. This behavior can be achieved using MPI\_COMM\_DUP\_WITH\_INFO and MPI\_COMM\_IDUP\_WITH\_INFO.

The default communicator where errors are raised when not involving a communicator, window, or file was changed from  $MPI_COMM_WORLD$  to  $MPI_COMM_SELF$ .

### 18.2 Additional Warnings

This section describes additional changes that could potentially cause a program that relies on the semantics described in a previous version of the MPI Standard to behave differently than with this version of MPI. The changes in this section are limited in scope and unlikely to impact most programs.

#### 18.2.1 Warnings Starting in MPI-4.0

The limit for length of MPI identifiers was removed. Prior to MPI-4.0, MPI identifiers were limited to 30 characters (31 with the profiling interface). This limitation was initially introduced to avoid exceeding the limit on some compilation systems.

Rationale. For Fortran, this limit was already relaxed for the Fortran specific function names, see Section 19.1.5, and the Fortran language specification 2003 requires support for a minimum of 63 characters for internal and external identifiers. Starting with the ISO/IEC 9899:1999 C programming language standard, support for a minimum of 63 characters is required for internal identifiers, but only 31 characters are required to be significant for external identifiers. At the time of the release of MPI-4.0, most or nearly all compilers allow external identifiers longer than 31 characters. Therefore, the restriction is removed. (End of rationale.)

Advice to users. This affects users only if they store MPI identifiers into fixed sized strings. (End of advice to users.)

 $\mathbf{2}$ 

 $\mathbf{5}$ 

## Chapter 19

# Language Bindings

### 19.1 Support for Fortran

#### 19.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [45] + TS 29113 [46].

Rationale. Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the ASYNCHRONOUS attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 19.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

- 1. USE mpi\_f08: This method is described in Section 19.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.
- 2. USE mpi: This method is described in Section 19.1.3 and requires compile-time argument checking. Handles are defined as INTEGER. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.
- 3. **INCLUDE 'mpif.h':** This method is described in Section 19.1.4. The use of the include file mpif.h is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

 $47 \\ 48$ 

1 MPI implementations providing a Fortran interface must provide one or both of the  $\mathbf{2}$ following: 3 • The USE mpi\_f08 Fortran support method. 4 5• The USE mpi and INCLUDE 'mpif.h' Fortran support methods. 6  $\overline{7}$ Section 19.1.6 describes restrictions if the compiler does not support all the needed features. 8 Application subroutines and functions may use either one of the modules or the mpif.h 9 include file. An implementation may require the use of one of the modules to prevent type 10mismatch errors. 11Advice to users. Users are advised to utilize one of the MPI modules even if mpif.h 12enforces type checking on a particular system. Using a module provides several poten-13 tial advantages over using an include file; the mpi\_f08 module offers the most robust 14and complete Fortran support. (End of advice to users.) 1516In a single application, it must be possible to link together routines which USE mpi\_f08, 17 USE mpi, and INCLUDE 'mpif.h'. 18 The LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED is set to .TRUE. if 19all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-20rank [46]; otherwise it is set to .FALSE.. The LOGICAL compile-time constant 21MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if the ASYNCHRONOUS attribute was 22added to the choice buffer arguments of all nonblocking interfaces and the underlying 23Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of  $^{24}$ TS 29113), otherwise it is set to .FALSE.. These constants exist for each Fortran support 25method, but not in the C header file. The values may be different for each Fortran support 26method. All other constants and the integer values of handles must be the same for each 27Fortran support method. 28Section 19.1.2 through 19.1.4 define the Fortran support methods. The Fortran in-29terfaces of each MPI routine are shorthands. Section 19.1.5 defines the corresponding full 30 interface specification together with the specific procedure names and implications for the  $^{31}$ profiling interface. Section 19.1.6 describes the implementation of the MPI routines for dif-32 ferent versions of the Fortran standard. Section 19.1.7 summarizes major requirements for 33 MPI implementations with Fortran support. Section 19.1.8 and Section 19.1.9 describe ad-34ditional functionality that is part of the Fortran support. MPI\_F\_SYNC\_REG is needed 35 for one of the methods to prevent register optimization problems. A set of functions 36 provides additional support for Fortran intrinsic numeric types, including parameterized 37 types: MPI\_TYPE\_MATCH\_SIZE, MPI\_TYPE\_CREATE\_F90\_INTEGER, 38 MPI\_TYPE\_CREATE\_F90\_REAL and MPI\_TYPE\_CREATE\_F90\_COMPLEX. In the context 39 of MPI, parameterized types are Fortran intrinsic types which are specified using KIND type 40 parameters. Sections 19.1.10 through 19.1.19 give an overview and details on known prob-41 lems when using Fortran together with MPI; Section 19.1.20 compares the Fortran problems 42with those in C. 43 44Fortran Support Through the mpi\_f08 Module 19.1.2 45

An MPI implementation providing a Fortran interface must provide a module named mpi\_f08
 that can be used in a Fortran program. Section 19.1.6 describes restrictions if the compiler
 does not support all the needed features. Within all MPI function specifications, the first

of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not TYPE(\*), with the following exception:

Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the mpi module, see Section 19.1.3.

Advice to users. It is strongly recommended that developers substitute calls to deprecated routines when upgrading from mpif.h or the mpi module to the mpi\_f08 module. (End of advice to users.)

- Define the derived type MPI\_Status, and define all MPI handles with uniquely named handle types (instead of INTEGER handles, as in the mpi module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators .EQ. and .NE. to allow the comparison of these MPI handles with .EQ., .NE., == and /=.
- Use the ASYNCHRONOUS attribute to protect the buffers of nonblocking operations, and set the LOGICAL compile-time constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING to .TRUE. if the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113). See Section 19.1.6 for older compiler versions.
- Set the LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED to .TRUE. and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., TYPE(\*), DIMENSION(..) in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, noncontiguous sub-arrays can be used as buffers in nonblocking routines.

*Rationale.* In all blocking routines, i.e., if the choice-buffer is not declared as ASYNCHRONOUS, the TS 29113 feature is not needed for the support of noncontiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI\_SUBARRAYS\_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of noncontiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 19.1.6 for details.
- Declare each argument with an INTENT of IN, OUT, or INOUT as defined in this standard.

 $\frac{4}{5}$ 

 $^{31}$ 

Rationale. For these definitions in the mpi\_f08 bindings, in most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for OUT and INOUT dummy arguments that allow one of the nonordinary Fortran constants (see MPI\_BOTTOM, etc. in Section 2.5.4) as input, an INTENT is not specified. (End of rationale.)

Advice to users. If a dummy argument is declared with INTENT(OUT), then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [45], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of INTENT(OUT) the copy-in may be suppressed by the optimization and the routine starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 19.1.3. (End of advice to users.)

 Declare all ierror output arguments as OPTIONAL, except for user-defined callback functions (e.g., of type MPI\_Comm\_copy\_attr\_function or COMM\_COPY\_ATTR\_FUNCTION) and predefined callbacks (e.g., MPI\_COMM\_NULL\_COPY\_FN).

 Rationale. For user-defined callback functions (e.g., of type
 MPI\_Comm\_copy\_attr\_function or COMM\_COPY\_ATTR\_FUNCTION) and their predefined callbacks (e.g., MPI\_COMM\_NULL\_COPY\_FN), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (End of rationale.)

The MPI Fortran bindings in the mpi\_f08 module are designed based on the Fortran 2008 standard [45] together with the Technical Specification "TS 29113 Further Interoperability with C" [46] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

Rationale. The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [46], "an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn."

The TS 29113 contains the following language features that are needed for the MPI bindings in the mpi\_f08 module: assumed-type and assumed-rank. It is important

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18 19

20

21

22

23

24 25

26

27

28

29

30

 $^{31}$ 

32

37

38

39

40

41

42

43

44

45

that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., REAL, CHARACTER\*5, CHARACTER\*(\*), sequence derived types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with mpif.h without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the mpi\_f08 module).

A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

#### 19.1.3 Fortran Support Through the mpi Module

An MPI implementation providing a Fortran interface must provide a module named mpi that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. If an implementation is paired with a compiler that either does not support TYPE(\*), DIMENSION(..) from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section 19.1.6 for more details.
- Define all MPI handles as type INTEGER.
- Define the derived type MPI\_Status and all named handle types that are used in the mpi\_f08 module. For these named handle types, overload the operators .EQ. and .NE. to allow handle comparison via the .EQ., .NE., == and /= operators.

*Rationale.* They are needed only when the application converts old-style INTEGER handles into new-style handles with a named type. (*End of rationale.*)

- A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the mpi\_f08 module if it is supported by the underlying compiler.
- Set the LOGICAL compile-time constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING to
   .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces and the
   underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113), otherwise to .FALSE..

 $\mathbf{2}$ 

For an MPI implementation that fully supports nonblocking calls Advice to users. with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses "contiguous" but not "simply contiguous" ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another reason may be that the application does not conform either to the MPI standard or to the Fortran standard, typically because the program forces the compiler to perform copyin/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is nonconforming. See Section 19.1.12 for more details. (End of advice to users.)

- A high quality MPI implementation may enhance the interface by using TYPE(\*), DIMENSION(..) choice buffer dummy arguments instead of using nonstandardized extensions such as !\$PRAGMA IGNORE\_TKR or a set of overloaded functions as described by M. Hennecke in [32], if the compiler supports this TS 29113 language feature. See Section 19.1.6 for further details.
  - Set the LOGICAL compile-time constant MPI\_SUBARRAYS\_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(\*), DIMENSION(..), otherwise set it to .FALSE.. When MPI\_SUBARRAYS\_SUPPORTED is defined as .TRUE., noncontiguous sub-arrays can be used as buffers in nonblocking routines.
  - Set the MPI\_SUBARRAYS\_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of noncontiguous sub-arrays in nonblock-ing calls may be disallowed. See Section 19.1.6 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi\_f08 bindings. (*End of advice to implementors.*)

Rationale. The intent given by the MPI generic interface is not precisely defined
 and does not in all cases correspond to the correct Fortran INTENT. For instance,
 receiving into a buffer specified by a datatype with absolute addresses may require
 associating MPI\_BOTTOM with a dummy OUT argument. Moreover, "constants" such
 MPI\_BOTTOM and MPI\_STATUS\_IGNORE are not constants as defined by Fortran,
 but "special addresses" used in a nonstandard way. Finally, the MPI-1 generic intent

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21 22

23

24

25

26

27 28

29

30

31

32

33 34

35

36

37 38

39

40

41

was changed in several places in MPI-2. For instance, MPI\_IN\_PLACE changes the intent of an OUT argument to be INOUT. (End of rationale.)

Advice to implementors. The Fortran 2008 standard illustrates in its Note 5.17 that "INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than INTENT(OUT), even if there is no explicit reference to the value of the dummy argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the IN-TENT attribute, because INTENT(INOUT) always requires that the associated actual argument is definable." Applications that include mpif.h may not expect that INTENT (OUT) is used. In particular, output array arguments are expected to keep their 12content as long as the MPI routine does not modify them. To keep this behavior, it is recommended that implementations not use INTENT(OUT) in the mpi module and the 14mpif.h include file, even though INTENT(OUT) is specified in an interface description of the mpi\_f08 module. (End of advice to implementors.)

#### Fortran Support Through the mpif.h Include File 19.1.4

The use of the mpif.h include file is strongly discouraged and may be deprecated in a future version of MPI.

An MPI implementation providing a Fortran interface must provide an include file named mpif.h that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:

• Define all named MPI constants.							
• Declare MPI functions that return a value.							
• Define all handles as INTEGER.							
• Be valid and equivalent for both fixed and free source form.							
For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted).							
• Set the LOGICAL compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING according to the same rules as for the mpi module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to .FALSE							
Advice to users. Instead of using mpif.h, the use of the mpi_f08 or mpi module is strongly encouraged for the following reasons:							
<ul> <li>Most mpif.h implementations do not include compile-time argument checking.</li> <li>Therefore, many bugs in MPI applications remain undetected at compile-time, such as:</li> </ul>							

- Missing ierror as last argument in most Fortran bindings.

797

1  $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8

9

10

11

13

15

1617 18

19

20

21

22

23

 $^{24}$ 

34

	198 CHAITER 19. LANGUA	AGE DINDINGS
1 2	<ul> <li>Declaration of a status as an INTEGER variable instead of a with size MPI_STATUS_SIZE.</li> </ul>	n INTEGER array
$\frac{3}{4}$	- Incorrect argument positions; e.g., interchanging the coun	t and
5	<ul><li>datatype arguments.</li><li>– Passing incorrect MPI handles; e.g., passing a datatype inst</li></ul>	ead of a commu-
6	nicator.	cad of a commu
7 8	• The migration from mpif.h to the mpi module should be relat	ively straightfor-
9 10	ward (i.e., substituting include 'mpif.h' after an implicit s mpi before that implicit statement) as long as the application s	-
11	• Migrating portable and correctly written applications to the m	i module is not
12 13	expected to be difficult. No compile or runtime problems shou an mpif.h include file was always allowed to provide explicit Fo	
14 15	(End of advice to users.)	
16	Rationale. The mpif.h include file has not been deprecated in order	to retain strong
17 18	backward compatibility. Internally, mpif.h and the mpi module may	-
19	so that essentially the same library implementation of the MPI routi	nes can be used.
20	(End of rationale.)	
21		c
22	19.1.5 Interface Specifications, Procedure Names, and the Profiling Int	ertace
23	The Fortran interface specification of each MPI routine specifies the routine	e name that must
24	be called by the application program, and the names and types of the du	
25 26	together with additional attributes. The Fortran standard allows a given I	
27	to be implemented with several methods, e.g., within or outside of a module	
28	BIND(C), or the buffers with or without TS 29113. Such implementation different binary interfaces and different specific procedure names. The	
29	several implementation schemes together with the rules for the specific p	-
30	and its implications for the profiling interface are specified within this sect	
31	implementation details.	,
32		
33 34	<i>Rationale.</i> When this section was originally introduced in MPI-3.0, for the three Fortran support methods were:	the major goals
35	• Portable implementation of the wrappers from the MPI Fortran	interfaces to the
36 37	MPI routines in C.	
38	• Binary backward compatible implementation path when swite	hing
39	MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE	0
40	$\bullet$ The Fortran $PMPI$ interface need not be backward compatible	e, but a method
41 42	must be included that a tools layer can use to examine the M the specific procedure names and interfaces used.	PI library about
43	• No performance drawbacks.	
44 45	• Consistency between all three Fortran support methods.	
46	• Consistent with Fortran 2008 + TS 29113.	
47		
48		

CHAPTER 19. LANGUAGE BINDINGS

No.	Specific pro- cedure name	Calling convention
1A	MPI_Isend_f08	Fortran interface and arguments, as in Annex A.4, except
		that in routines with a choice buffer dummy argument,
		this dummy argument is implemented with nonstandard ex-
		tensions like <b>!</b> \$PRAGMA IGNORE_TKR, which provides a call-
		by-reference argument without type, kind, and dimension checking.
В	MPI_Isend_f08ts	Fortran interface and arguments, as in Annex A.4, but
		only for routines with one or more choice buffer dummy
		arguments; these dummy arguments are implemented with
		TYPE(*), DIMENSION().
2A	MPI_ISEND	Fortran interface and arguments, as in Annex A.5, except
		that in routines with a choice buffer dummy argument,
		this dummy argument is implemented with nonstandard ex-
		tensions like <b>!</b> \$PRAGMA IGNORE_TKR, which provides a call-
		by-reference argument without type, kind, and dimension
		checking.
$2\mathrm{B}$	MPI_ISEND_FTS	Fortran interface and arguments, as in Annex A.5, but
		only for routines with one or more choice buffer dummy
		arguments; these dummy arguments are implemented
		with TYPE(*), DIMENSION(). In mpif.h only, the
		postfix "_FTS" for MPI_NEIGHBOR_ALLGATHERV_INIT,
		MPI_NEIGHBOR_ALLTOALLV_INIT, and
		MPI_NEIGHBOR_ALLTOALLW_INIT is shortened to "_F".

Table 19.1: Specific Fortran procedure names and related calling conventions. MPI\_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was not fulfilled. The LOGICAL arguments are not interoperable with C, mainly because the internal representations for .FALSE. and .TRUE. are compiler dependent. The provided interface was mainly based on BIND(C) interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the BIND(C) had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI\_COMM\_DUP\_FN. Non-BIND(C) procedures are also not interoperable with C, and therefore the BIND(C) had to be removed from all routines with PROCEDURE arguments, e.g., from MPI\_OP\_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (End of rationale.)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 19.1. Case is not significant in the names.

Note that for the deprecated routines in Section 16.1, which are reported only in Annex A.5, scheme 2A is utilized in the mpi module and mpif.h, and also in the mpi\_f08 module.

 $^{31}$ 

To set MPI\_SUBARRAYS\_SUPPORTED to .TRUE. within a Fortran support method, it is required that all nonblocking and split-collective routines with buffer arguments are implemented according to 1B and 2B, i.e., with MPI\_Xxxx\_f08ts in the mpi\_f08 module, and with MPI\_XXXX\_FTS in the mpi module and the mpif.h include file.

<sup>5</sup> The mpi and mpi\_f08 modules and the mpif.h include file will each correspond to <sup>6</sup> exactly one implementation scheme from Table 19.1. However, the MPI library may contain <sup>7</sup> multiple implementation schemes from Table 19.1.

Advice to implementors. This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)

12After a compiler provides the facilities from TS 29113, i.e., TYPE(\*), Rationale. 13 DIMENSION(...), it is possible to change the bindings within a Fortran support method 14to support subarrays without recompiling the complete application provided that the 15previous interfaces with their specific procedure names are still included in the li-16brary. Of course, only recompiled routines can benefit from the added facilities. 17 There is no binary compatibility conflict because each interface uses its own spe-18 cific procedure names and all interfaces use the same constants (except the value of 19 MPI\_SUBARRAYS\_SUPPORTED and MPI\_ASYNC\_PROTECTS\_NONBLOCKING) and type 20definitions. After a compiler also ensures that buffer arguments of nonblocking MPI 21operations can be protected through the ASYNCHRONOUS attribute, and the proce-22 dure declarations in the mpi\_f08 and mpi module and the mpif.h include file declare 23choice buffers with the ASYNCHRONOUS attribute, then the value of 24

- MPI\_ASYNC\_PROTECTS\_NONBLOCKING can be switched to .TRUE. in the module definition and include file. (*End of rationale.*)
  - Advice to users. Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is—and what is not—supported. (*End of advice to users.*)

Within the mpi\_f08 and mpi modules and mpif.h, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI\_Isend. The specific procedure names for these PMPI\_Xxxx procedures must be different from the specific procedure names for the MPI\_Xxxx procedures and are not specified by this standard.

<sup>37</sup> A user-written or middleware profiling routine should provide the same specific Fortran
 <sup>38</sup> procedure names and calling conventions, and therefore can interpose itself as the MPI
 <sup>39</sup> library routine. The profiling routine can internally call the matching

PMPI routine with any of its existing bindings, except for routines that have callback routine
 dummy arguments, choice buffer arguments, or that are attribute caching routines (

<sup>42</sup> MPI\_{COMM|WIN|TYPE}\_{SET|GET}\_ATTR). In this case, the profiling software should <sup>43</sup> invoke the corresponding PMPI routine using the same Fortran support method as used in <sup>44</sup> the calling application program, because the C, mpi\_f08 and mpi callback prototypes are <sup>45</sup> different or the meaning of the choice buffer or attribute\_val arguments are different.

 $46 \\ 47$ 

Advice to users. Although for each support method and MPI routine (e.g.,
 MPI\_ISEND in mpi\_f08), multiple routines may need to be provided to intercept

1

 $\mathbf{2}$ 

3

4

8

9

10

11

27

28

29

30

 $^{31}$ 

the specific procedures in the MPI library (e.g., MPI\_lsend\_f08 and MPI\_lsend\_f08ts), each profiling routine itself uses only one support method (e.g., mpi\_f08) and calls the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI\_lsend in this example). (*End of advice to users.*)

Advice to implementors. If all of the following conditions are fulfilled:

- the handles in the mpi\_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a nonoptional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

Advice to implementors. In the Fortran support method mpif.h, compile-time argument checking can be also implemented for all routines. For mpif.h, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that mpif.h must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the three longest subroutine declaration statements are

```
SUBROUTINE PMPI_DIST_GRAPH_CREATE_ADJACENT(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_NEIGHBOR_ALLTOALLW_INIT(a,b,c,d,e,f,g,h,i,j,k,l)
SUBROUTINE PMPI_NEIGHBOR_ALLTOALLV_INIT(a,b,c,d,e,f,g,h,i,j,k,l)
```

with 71 and 70 characters each. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. Some of the longest of such interface definitions are

41
42
43
43
44
45
46
47

 $^{31}$ 

1	with 72, 71, and 70 characters. In principle, continuation lines would be possible							
2	in mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the							
3	continuation line) but this would not be valid if the source line length is extended							
4	with a compiler flag to 132 characters. Column 133 is also not available for the							
5	continuation character because lines longer than 132 characters are invalid with some							
6	compilers by default.							
7	The longest specific procedure name is PMPI_Reduce_scatter_block_init_c_f08ts with							
8	38 characters in the mpi_f08 module.							
9	-							
10	For example, the interface specifications together with the specific procedure names can be implemented with							
11	can be implemented with							
12	MODULE mpi_f08							
13	TYPE, BIND(C) :: MPI_Comm							
14	INTEGER :: MPI_VAL							
15	END TYPE MPI_Comm							
16	 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)							
17	SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)							
18	IMPORT :: MPI_Comm							
19	TYPE(MPI_Comm), INTENT(IN) :: comm							
20	INTEGER, INTENT(OUT) :: rank							
21	INTEGER, OPTIONAL, INTENT(OUT) :: ierror END SUBROUTINE							
22 23	END SUBRUCTINE END INTERFACE							
23	END MODULE mpi_f08							
25	-							
26	MODULE mpi							
27	INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)							
28	SUBROUTINE MPI_Comm_rank(comm, rank, ierror) INTEGER, INTENT(IN) :: comm ! The INTENT may be added although							
29	INTEGER, INTENT(OUT) :: rank ! it is not defined in the							
30	INTEGER, INTENT(OUT) :: ierror ! official routine definition.							
31	END SUBROUTINE							
32	END INTERFACE							
33	END MODULE mpi							
34	And if interfaces are provided in mpif.h, they might look like this (outside of any							
35	module and in fixed source format):							
36								
37	!23456789012345678901234567890123456789012345678901234567890123456789012 INTERFACE MPI_Comm_rank ! (as defined in Chapter 6)							
38	SUBROUTINE MPI_Comm_rank(comm, rank, ierror)							
39	INTEGER, INTENT(IN) :: comm ! The argument names may be							
40	INTEGER, INTENT(OUT) :: rank ! shortened so that the							
41	INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the							
42	END SUBROUTINE ! maximum of 72 characters.							
43	END INTERFACE							
44	(End of advice to implementors.)							
45								
46	Advice to users. The following is an example of how a user-written or middleware							
47	profiling routine can be implemented:							
48								

SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)							
USE :: mpi_f08, my_noname => MPI_Isend_f08ts							
TYPE(*), DIMENSION(), ASYNCHRONOUS :: h	buf						
INTEGER, INTENT(IN) :: c	count, dest, tag						
<pre>TYPE(MPI_Datatype), INTENT(IN) :: d</pre>	datatype						
TYPE(MPI_Comm), INTENT(IN) :: c	comm						
TYPE(MPI_Request), INTENT(OUT) :: r	request						
INTEGER, OPTIONAL, INTENT(OUT) :: i	ierror						
! some code for the begin of profil	ling						
call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)							
! some code for the end of profilir	ng						
END SUBROUTINE MPI_Isend_f08ts							

Note that this routine is used to intercept the existing specific procedure name MPI\_lsend\_f08ts in the MPI library. This routine must not be part of a module. This routine itself calls PMPI\_lsend. The USE of the mpi\_f08 module is needed for definitions of handle types and the interface for PMPI\_lsend. However, this module also contains an interface definition for the specific procedure name MPI\_lsend\_f08ts that conflicts with the definition of this profiling routine (i.e., the name is doubly defined). Therefore, the USE here specifically excludes the interface from the module by renaming the unused routine name in the mpi\_f08 module into "my\_noname" in the scope of this routine. (*End of advice to users.*)

The PMPI interface allows intercepting MPI routines. For exam-Advice to users. ple, an additional MPI\_ISEND profiling wrapper can be provided that is called by the application and internally calls PMPI\_ISEND. There are two typical use cases: a profiling layer that is developed independently from the application and the MPI library, and profiling routines that are part of the application and have access to the application data. With MPI-3.0, new Fortran interfaces and implementation schemes were introduced that have several implications on how Fortran MPI routines are internally implemented and optimized. For profiling layers, these schemes imply that several internal interfaces with different specific procedure names may need to be intercepted, as shown in the example code above. Therefore, for wrapper routines that are part of a Fortran application, it may be more convenient to make the name shift within the application, i.e., to substitute the call to the MPI routine (e.g., MPI\_ISEND) by a call to a user-written profiling wrapper with a new name (e.g., X\_MPI\_ISEND) and to call the Fortran MPI\_ISEND from this wrapper, instead of using the PMPI interface. (End of advice to users.)

Advice to implementors. An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 19.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

#### 19.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

• For Fortran 77 with some extensions:

 $^{24}$ 

 $46 \\ 47$ 

1	- MPI identifiers may be up to 30 characters (31 with the profiling interface).
2	– MPI identifiers may contain underscores after the first character.
3	- An MPI subroutine with a choice argument may be called with different argument
4 5	types.
6	- Although not required by the MPI standard, the INCLUDE statement should be
7	available for including mpif.h into the user application source code.
8	Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute ad-
9 10	dresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address
11 12	does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with MPI_GET_ADDRESS, but not for Fortran 77.)
13	- For Fortman 00:
14	• For Fortran 90: The major additional features that are needed from Fortran 90 are:
15	
16	- The MODULE and INTERFACE concept.
17	- The KIND= and SELECTED_XXX_KIND concept.
18 19	- Fortran derived TYPEs and the SEQUENCE attribute.
20	- The OPTIONAL attribute for dummy arguments.
21	- Cray pointers, which are a nonstandard compiler extension, are needed for the
22	use of MPI_ALLOC_MEM.
23	With these features, $MPI-1.1 - MPI-2.2$ can be implemented without restrictions.
24 25	MPI-3.0 and later can be implemented with some restrictions. The Fortran support
26	methods are abbreviated with $S1 = \text{the mpi_f08}$ module, $S2 = \text{the mpi}$ module, and
27	S3 = the mpif.f include file. If not stated otherwise, restrictions exist for each method
28	that prevent implementing the complete semantics of MPI.
29	MDI SUBADDAYS SUDDODTED aguala FAISE i.e. subscript triplate and non
30	<ul> <li>MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non- contiguous subarrays cannot be used as buffers in nonblocking routines, RMA,</li> </ul>
31 32	or split-collective I/O.
33	- S1, S2, and S3 can be implemented, but for S1, only a preliminary implementa-
34	tion is possible.
35	- In this preliminary interface of S1, the following changes are necessary:
36	* TYPE(*), DIMENSION() is substituted by nonstandardized extensions like
37	* THE (*), DIMENSION() IS SUBSTITUTED BY HOLSTANDARDIZED EXTENSIONS INC !\$PRAGMA_IGNORE_TKR.
38 39	* The ASYNCHRONOUS attribute is omitted.
40	* PROCEDURE() callback declarations are substituted by EXTERNAL.
41	- The specific procedure names are specified in Section 19.1.5.
42	<ul> <li>Due to the rules specified in Section 19.1.5, choice buffer declarations should be</li> </ul>
43	implemented only with nonstandardized extensions like <b>!\$PRAGMA IGNORE_TKR</b>
44	(as long as F2008+TS 29113 is not available).
45	In S2 and S3: Without such extensions, routines with choice buffers should be
46 47	provided with an implicit interface, instead of overloading with a different MPI
48	function for each possible buffer type (as mentioned in Section 19.1.11). Such

overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 19.1.15.

Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommended not to provide the  $mpi_f08$  module if such an extension is not available.

- The ASYNCHRONOUS attribute can not be used in applications to protect buffers in nonblocking MPI calls (S1–S3).
- The TYPE(C\_PTR) binding of the MPI\_ALLOC\_MEM and MPI\_WIN\_ALLOCATE routines is not available.
- In S1 and S2, the definition of the handle types (e.g., TYPE(MPI\_Comm) and the status type TYPE(MPI\_Status) must be modified: The SEQUENCE attribute must be used instead of BIND(C) (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the mpi\_f08 module for Fortran 90 compilers. In this case, the mpi\_f08 handle types and all routines, constants and types related to TYPE(MPI\_Status) (see Section 19.3.5) are also not available in the mpi module and mpif.h.

•	For Fortran 95:	22						
	The quality of the MPI interface and the restrictions are the same as with Fortran 90.	23						
		24						
•	For Fortran 2003:	25						
	The major features that are needed from Fortran 2003 are:	26						
	- Interoperability with C, i.e.,	27						
	* BIND(C) derived types.	28						
	· *	29						
	* The ISO_C_BINDING intrinsic type C_PTR and routine C_F_POINTER.	30						
	- The ability to define an <code>ABSTRACT INTERFACE</code> and to use it for <code>PROCEDURE</code> dummy	31						
	arguments.	32						
	– The ability to overload the operators .EQ. and .NE. to allow the comparison of	33						
	derived types (used in MPI-3.0 and later for MPI handles).	34						
	- The ASYNCHRONOUS attribute is available to protect Fortran asynchronous I/O.	35 36						
	This feature is not yet used by MPI, but it is the basis for the enhancement for							
	MPI communication in the TS 29113.							
		38						
	With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2	39						
	can be implemented without restrictions, but with one enhancement:	40						
		41						
	- The user application can use TYPE(C_PTR) together with MPI_ALLOC_MEM as	42						
	long as MPI_ALLOC_MEM is defined with an implicit interface because a C_PTR and an INTEGER(KIND=MPI_ADDRESS_KIND) argument must both map to a							
	void * argument.	45						
	MPI-3.0 and later can be implemented with the following restrictions:							
	with the following restrictions.	47						
	— MPI_SUBARRAYS_SUPPORTED equals .FALSE	48						

1 2

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10

11 12

13

14

15 16

17

18

19 20

1	$-$ For ${\sf S1},$ only a preliminary implementation is possible. The following changes are
2	necessary:
3	* TYPE(*), DIMENSION() is substituted by nonstandardized extensions like
4	!\$PRAGMA IGNORE_TKR.
5	- The specific procedure names are specified in Section 19.1.5.
6	
7	- With S1, the ASYNCHRONOUS is required as specified in the second Fortran inter-
8	faces. With S2 and S3 the implementation can also add this attribute if explicit
9	interfaces are used.
10	- The ASYNCHRONOUS Fortran attribute can be used in applications to try to protect
11	buffers in nonblocking MPI calls, but the protection can work only if the compiler
12	is able to protect asynchronous Fortran $I/O$ and makes no difference between such
13	asynchronous Fortran $I/O$ and MPI communication.
14	- The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
15	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
16	be used only for Fortran types that are C compatible.
17	- The same restriction as for Fortran 90 applies if nonstandardized extensions like
18	<ul> <li>The same restriction as for Fortran 90 applies if nonstandardized extensions like</li> <li>!\$PRAGMA_IGNORE_TKR are not available.</li> </ul>
19	! \$FRAGMA IGNORE_IRR are not available.
20	• For Fortran $2008 + TS 29113$ and later and
21	For Fortran 2003 + TS 29113:
22	The major features that are needed from TS 29113 are:
23	
24 25	- TYPE(*), DIMENSION() is available.
26	- The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI com-
27	munication.
28	- The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not
29	restricted to Fortran types for which a corresponding type in C exists.
30	
31	Using these features, MPI-3.0 and later can be implemented without any restrictions.
32	$-$ With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE The
33	ASYNCHRONOUS attribute can be used to protect buffers in nonblocking MPI calls.
34	The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE,
35	MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can
36	be used for any Fortran type.
37	
38	- With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation
39	dependent. A high quality implementation will also provide
40	MPI_SUBARRAYS_SUPPORTED set to .TRUE. and will use the ASYNCHRONOUS at-
41	tribute in the same way as in S1.
42	$-$ If nonstandardized extensions like $\tt !\$PRAGMA IGNORE\_TKR$ are not available then
43	S2 must be implemented with TYPE(*), DIMENSION().
44	
45	Advice to implementors. If $MPI_SUBARRAYS_SUPPORTED ==.FALSE.$ , the choice
46	argument may be implemented with an explicit interface using compiler directives,
47	for example:
48	

```
INTERFACE
SUBROUTINE MPI_...(buf, ...)
!DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
!$PRAGMA IGNORE_TKR buf
!DIR$ IGNORE_TKR buf
!IBM* IGNORE_TKR buf
REAL, DIMENSION(*) :: buf
... ! declarations of the other arguments
END SUBROUTINE
END INTERFACE
```

(End of advice to implementors.)

#### 19.1.7 Requirements on Fortran Compilers

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

Advice to users. Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., mpif90, mpicc) that set these options automatically. (End of advice to users.)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI\_GET\_VERSION, if all the solutions described in Sections 19.1.11 through 19.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the mpi\_f08 and mpi modules, and mpif.h) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [46] are available. This is required only for mpi\_f08. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the mpi\_f08 module with MPI\_SUBARRAYS\_SUPPORTED set to .FALSE..
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the ASYNCHRONOUS attribute. See Section 19.1.12 for more details.
- SEQUENCE and BIND(C) derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI\_SUBARRAYS\_SUPPORTED== .FALSE., they are passed with call by reference, and passed by descriptor in the case of .TRUE..
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., CHARACTER(LEN=\*) strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the ISO\_C\_BINDING intrinsic module procedure C\_F\_POINTER is not restricted to Fortran types for which a corresponding type in C exists.

 $^{24}$ 

 $45 \\ 46$ 

• The Fortran compiler shall not provide TYPE(\*) unless the ASYNCHRONOUS attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the ASYNCHRONOUS attribute as part of TS 29113 and there still exists a Fortran 6 support method with MPI\_ASYNC\_PROTECTS\_NONBLOCKING set to .FALSE.. Observation of these rules by the MPI application developer is especially recommended for backward compatibility of existing applications that use the mpi module or the mpif.h include file. The rules are as follows: 10

- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (e.g., MPI\_F\_SYNC\_REG on page 830 and Section 19.1.8, and DD on page 831) solve the problems described in Section 19.1.17.
- The problems with temporary data movement (described in detail in Section 19.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.
  - Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 19.1.19) are resolved without any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.

All of these rules are valid for the mpi\_f08 and mpi modules and independently of whether mpif.h uses explicit interfaces.

Advice to implementors. Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [46], and some of these requirements for MPI are beyond the scope of TS 29113. (End of advice to *implementors.*)

19.1.8 Additional Support for Fortran Register-Memory-Synchronization

34As described in Section 19.1.17, a dummy call may be necessary to tell the compiler that 35 registers are to be flushed for a given buffer or that accesses to a buffer may not be moved 36 across a given point in the execution sequence. Only a Fortran binding exists for this call. 37

```
38
39
     MPI_F_SYNC_REG(buf)
40
       INOUT
                 buf
                                             initial address of buffer (choice)
41
42
     Fortran 2008 binding
43
     MPI_F_sync_reg(buf)
44
          TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
45
46
     Fortran binding
47
     MPI_F_SYNC_REG(BUF)
48
          <type> BUF(*)
```

1

 $\mathbf{2}$ 

3

4

5

7

8

9

11

12

13

14

15

16

17

18

19 20

21

22

23

 $^{24}$ 25

26

27

28

29

30

 $^{31}$ 32

This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

*Rationale.* This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

Advice to implementors. This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the mpi\_f08 module according to the definition for the mpi module or mpif.h to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (End of advice to implementors.)

Rationale. This routine is not defined with TYPE(\*), DIMENSION(\*), i.e., assumed size instead of assumed rank, because this would restrict the usability to "simply contiguous" arrays and would require overloading with another interface for scalar arguments. (End of rationale.)

Advice to users. If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to MPI\_F\_SYNC\_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not be called if MPI\_ASYNC\_PROTECTS\_NONBLOCKING is .TRUE. and the application fully uses the facilities of ASYNCHRONOUS arrays. (*End of advice to users*.)

#### 19.1.9 Additional Support for Fortran Numeric Intrinsic Types

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI\_INTEGER, MPI\_REAL, MPI\_INT, MPI\_DOUBLE, etc., as well as the optional types MPI\_REAL4, MPI\_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called KIND-parameterized types. These 35 types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL, and 36 CHARACTER) with an optional integer KIND parameter that selects from among one or more 37 variants. The specific meaning of different KIND values themselves are implementation 38 dependent and not specified by the language. Fortran provides the KIND selection functions 39 selected\_real\_kind for REAL and COMPLEX types, and selected\_int\_kind for INTEGER 40 types that allow users to declare variables with a minimum precision or number of digits. 41 These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX, and 42INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL 43 and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE 44PRECISION variables are of intrinsic type REAL with a nondefault KIND. The following two 45declarations are equivalent: 46

double precision x
real(KIND(0.0d0)) x

1

 $\mathbf{2}$ 

3

4

5

6

 $\overline{7}$ 

8 9

10

11

12

13

14

15

16 17

18

19

20

21

22

23

 $^{24}$ 

25

26

27 28

29 30

 $^{31}$ 

32

33

34

47

1 MPI provides two orthogonal methods for handling communication buffers of numeric  $\mathbf{2}$ intrinsic types. The first method (see the following section) can be used when variables have 3 been declared in a portable way—using default KIND or using KIND parameters obtained 4 with the selected\_int\_kind or selected\_real\_kind functions. With this method, MPI  $\mathbf{5}$ automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation 6 conversion in heterogeneous environments. The second method (see "Support for size- $\overline{7}$ specific MPI Datatypes" on page 814) gives the user complete control over communication 8 by exposing machine representations.

9 10

11

#### Parameterized Datatypes with Specified Precision and Exponent Range

<sup>12</sup> MPI provides named datatypes corresponding to standard Fortran 77 numeric types:

<sup>13</sup> MPI\_INTEGER, MPI\_COMPLEX, MPI\_REAL, MPI\_DOUBLE\_PRECISION and

<sup>14</sup> MPI\_DOUBLE\_COMPLEX. MPI automatically selects the correct data size and provides rep-<sup>15</sup> resentation conversion in heterogeneous environments. The mechanism described in this <sup>16</sup> section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables 17are declared (perhaps indirectly) using selected\_real\_kind(p, r) to determine the KIND 18 parameter, where  $\mathbf{p}$  is decimal digits of precision and  $\mathbf{r}$  is an exponent range. Implicitly 19MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is 20defined for each value of (p, r) supported by the compiler, including pairs for which one 21value is unspecified. Attempting to access an element of the array with an index (p, r) not 22 supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX 23datatypes. For integers, there is a similar implicit array related to **selected\_int\_kind** and 24indexed by the requested number of digits **r**. Note that the predefined datatypes contained 25in these implicit arrays are not the same as the named MPI datatypes MPI\_REAL, etc., but 26a new set. 27

Advice to implementors. The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (*End of advice to implementors.*)

Advice to users. selected\_real\_kind() maps a large number of (p,r) pairs to a much smaller number of KIND parameters supported by the compiler. KIND parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and KIND parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same (p,r) value (REAL and COMPLEX) or r value (INTEGER). Thus MPI has many more datatypes than there are fundamental language types. (End of advice to users.)

40 41

28

29

30

31 32

33

34

35

36

37

38

- 42
- 43
- 44 45
- 16
- 46
- 47 48

MPI_TYP	'E_CREATE_F90_REAL(p, r, n	ewtype)	1				
IN	р — — — (1777	precision, in decimal digits (integer)	2				
IN	r	decimal exponent range (integer)	3 4				
OUT	newtype	the requested MPI datatype (handle)	5				
	51		6				
C bindir	5		7 8				
int MPI_	Type_create_f90_real(int	p, int r, MPI_Datatype *newtype)	9				
	2008 binding		10				
	_create_f90_real(p, r, ne GER, INTENT(IN) :: p, r	ewtype, ierror)	11				
	(MPI_Datatype), INTENT(OU	JT) :: newtype	12 13				
	GER, OPTIONAL, INTENT(OUT	• -	14				
Fortran	binding		15				
	_CREATE_F90_REAL(P, R, NE	-	16 17				
INTE	GER P, R, NEWTYPE, IERROF	1	18				
	*	MPI datatype that matches a REAL variable of KIND	19				
	-	odel described above it returns a handle for the el- omitted from calls to selected_real_kind(p, r)	20 21				
		or <b>r</b> may be set to MPI_UNDEFINED. In communica-	21				
tion, an N	IPI datatype A returned by N	IPI_TYPE_CREATE_F90_REAL matches a datatype	23				
	0	PI_TYPE_CREATE_F90_REAL called with the same such a datatype. Restrictions on using the returned	24				
	-	resentation are given on page 813.	25 26				
• *	It is erroneous to supply values for $p$ and $r$ not supported by the compiler. <sup>27</sup>						
	28						
MPI_TYP	E_CREATE_F90_COMPLEX(	o, r, newtype)	29 30				
IN	р	precision, in decimal digits (integer)	31				
IN	r	decimal exponent range (integer)	32				
OUT	newtype	the requested MPI datatype (handle)	33 34				
			35				
C bindir	0		36				
int MPL_	Type_create_f90_complex(i	nt p, int r, MPI_Datatype *newtype)	37 38				
	2008 binding		39				
	<pre>_create_f90_complex(p, r, GER, INTENT(IN) :: p, r</pre>	, newtype, lerror)	40				
	(MPI_Datatype), INTENT(OU	JT) :: newtype	41				
INTE	GER, OPTIONAL, INTENT(OUT	T) :: ierror	42 43				
Fortran	_		44				
	_CREATE_F90_COMPLEX(P, R,		45				
LNIE	GER P, R, NEWTYPE, IERROF	ı	46 47				

1 2 3 4 5 6 7 8 9	This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 813. It is erroneous to supply values for p and r not supported by the compiler.							
10	MPI_TYPI	E_CREATE_F90_INTEGER(	(r, newtype)					
11 12	IN	r	decimal exponent range, i.e., number of decimal digits (integer)					
13 14	OUT	newtype	the requested MPI datatype (handle)					
15 16 17	C bindin int MPI_7	0	(int r, MPI_Datatype *newtype)					
18 19 20 21 22 23	<pre>Fortran 2008 binding MPI_Type_create_f90_integer(r, newtype, ierror) INTEGER, INTENT(IN) :: r TYPE(MPI_Datatype), INTENT(OUT) :: newtype INTEGER, OPTIONAL, INTENT(OUT) :: ierror</pre>							
23 24 25 26		Dinding _CREATE_F90_INTEGER(R, GER R, NEWTYPE, IERROR	NEWTYPE, IERROR)					
27 28 29 30 31 32 33	KIND sele analogous Restriction given on p	ected_int_kind(r). Matc to the matching rules for d as on using the returned d age 813. erroneous to supply a value	ed MPI datatype that matches an INTEGER variable of hing rules for datatypes created by this function are atatypes created by MPI_TYPE_CREATE_F90_REAL. atatype with the "external32" data representation are for r that is not supported by the compiler.					
34	Dam	pie.						
35	integer	longtype, quadtyp						
36	-	parameter :: long = se	lected_int_kind(15)					
37	•	long) ii(10) ected_real_kind(30)) x(	10)					
38			R(15, longtype, ierror)					
39 40			0, MPI_UNDEFINED, quadtype, ierror)					
40 41		_IIFE_CREATE_F90_REAL(3	o, MPI_ONDEFINED, quadtype, lellol)					
42	11 107		<b>`</b>					
43		SEND(ii, 10, longtype,						
44	Call MPL	SEND(x, 10, quadtype,	)					
45	A dava	<i>ice to users.</i> The dataty	vpes returned by the above functions are predefined					
46		U	ed; they do not need to be committed; they can be					
47 48			operations. There are two situations in which they					

behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

- 1. MPI\_TYPE\_GET\_ENVELOPE returns special combiners that allow a program to retrieve the values of **p** and **r**.
- 2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI\_TYPE\_CREATE\_F90\_XXX routines.

If a variable was declared specifying a nondefault KIND value that was not obtained with selected\_real\_kind() or selected\_int\_kind(), the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section. (*End of advice to users.*)

Advice to implementors. An application may often repeat a call to MPI\_TYPE\_CREATE\_F90\_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/ INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI\_TYPE\_CREATE\_F90\_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

*Rationale.* The MPI\_TYPE\_CREATE\_F90\_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 14.5.2) or user-defined (Section 14.5.3) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 14.5.2.

The "external32" representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The "external32" representations of the datatypes returned by MPI\_TYPE\_CREATE\_F90\_REAL/COMPLEX/INTEGER are given by the following rules. For MPI\_TYPE\_CREATE\_F90\_REAL:

if		(p	>	33)	or	(r	>	4931)	then	external32 representation
										is undefined
else i	if	(p	>	15)	or	(r	>	307)	then	external32_size = 16
else i	if	(p	>	6)	or	(r	>	37)	then	external32_size = 8
else										external32_size = 4

 $^{31}$ 

```
1
     For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for
\mathbf{2}
     MPI_TYPE_CREATE_F90_REAL.
3
     For MPI_TYPE_CREATE_F90_INTEGER:
4
                  (r > 38) then external32 representation is undefined
         if
5
         else if (r > 18) then
                                   external32_size =
                                                          16
6
         else if (r > 9) then
                                    external32_size =
                                                          8
7
         else if (r > 4) then
                                    external32_size =
                                                          4
8
         else if (r > 2) then
                                    external32_size =
                                                          2
9
                                    external32_size =
         else
                                                          1
10
     If the "external32" representation of a datatype is undefined, the result of using the datatype
11
     directly or indirectly (i.e., as part of another datatype or through a duplicated datatype)
12
     in operations that require the "external32" representation is undefined. These operations in-
13
     clude MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, and many MPI_FILE functions,
14
     when the "external32" data representation is used. The ranges for which the "external32"
15
16
     representation is undefined are reserved for future standardization.
17
18
     Support for Size-specific MPI Datatypes
19
     MPI provides named datatypes corresponding to optional Fortran 77 numeric types that
20
     contain explicit byte lengths—MPI_REAL4, MPI_INTEGER8, etc. This section describes a
21
     mechanism that generalizes this model to support all Fortran numeric intrinsic types.
22
          We assume that for each typeclass (integer, real, complex) and each word size there is
23
     a unique machine representation. For every pair (typeclass, n) supported by a compiler,
^{24}
     MPI must provide a named size-specific datatype. The name of this datatype is of the form
25
     MPI_<TYPE>n in C and Fortran where <TYPE> is one of REAL, INTEGER and COMPLEX, and
26
     n is the length in bytes of the machine representation. This datatype locally matches all
27
     variables of type (typeclass, n) in Fortran. The list of names for such types includes:
28
     MPI REAL4
29
30
     MPI_REAL8
^{31}
     MPI_REAL16
32
     MPI_COMPLEX8
33
     MPI_COMPLEX16
34
     MPI_COMPLEX32
35
     MPI_INTEGER1
36
     MPI_INTEGER2
37
     MPI_INTEGER4
38
     MPI_INTEGER8
39
     MPI_INTEGER16
40
     One datatype is required for each representation supported by the Fortran compiler.
41
42
           Rationale.
                       Particularly for the longer floating-point types, C and Fortran may use
43
           different representations. For example, a Fortran compiler may define a 16-byte REAL
44
           type with 33 decimal digits of precision while a C compiler may define a 16-byte long
45
           double type that implements an 80-bit (10 byte) extended precision floating point
46
           value. Both of these types are 16 bytes long, but they are not interoperable. Thus,
47
           these types are defined by Fortran, even though C may define types of the same length.
48
           (End of rationale.)
```

To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations REAL\*n, INTEGER\*n, always create a variable whose representation is of size n. These datatypes may also be used for variables declared with KIND=INT8/16/32/64 or KIND=REAL32/64/128, which are defined in the ISO\_FORTRAN\_ENV intrinsic module. Note that the MPI datatypes and the REAL\*n, INTEGER\*n declarations count bytes whereas the Fortran KIND values count bits. All these datatypes are predefined.

The following function allows a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

#### MPI\_TYPE\_MATCH\_SIZE(typeclass, size, datatype)

IN	typeclass	generic type specifier (integer)
IN	size	size, in bytes, of representation (integer)
OUT	datatype	datatype with correct type, size (handle)

#### C binding

int MPI\_Type\_match\_size(int typeclass, int size, MPI\_Datatype \*datatype)

#### Fortran 2008 binding

```
MPI_Type_match_size(typeclass, size, datatype, ierror)
    INTEGER, INTENT(IN) :: typeclass, size
    TYPE(MPI_Datatype), INTENT(OUT) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

#### Fortran binding

```
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

typeclass is one of MPI\_TYPECLASS\_REAL, MPI\_TYPECLASS\_INTEGER and MPI\_TYPECLASS\_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (**typeclass**, **size**).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI\_TYPE\_MATCH\_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling storage\_size() in order to compute the variable size in bits, dividing it by eight, and then calling MPI\_TYPE\_MATCH\_SIZE to find a suitable datatype. In C, one can use the C function sizeof() (which returns the size in bytes) instead of storage\_size() (which returns the size in bits). In addition, for variables of default kind the variable's size can be computed by a call to MPI\_TYPE\_GET\_EXTENT, if the typeclass is known. It is erroneous to specify a size not supported by the compiler.

*Rationale.* This is a convenience function. Without it, it can be tedious to find the correct named type. See note to implementors below. (*End of rationale.*)

Advice to implementors. This function could be implemented as a series of tests.

int MPI\_Type\_match\_size(int typeclass, int size, MPI\_Datatype \*rtype)
{

switch(typeclass) {

 $\overline{7}$ 

 $45 \\ 46$ 

```
1
                  case MPI_TYPECLASS_REAL: switch(size) {
2
                    case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
3
                    case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
4
                    default: error(...);
5
                  }
6
                  case MPI_TYPECLASS_INTEGER: switch(size) {
7
                     case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
8
                     case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
9
                     default: error(...);
10
                  }
11
                 ... etc. ...
12
              }
13
14
              return MPI_SUCCESS;
15
           }
16
           (End of advice to implementors.)
17
18
19
     Communication With Size-specific Types
20
     The usual type matching rules apply to size-specific datatypes: a value sent with datatype
21
     MPI_{<TYPE>n} can be received with this same datatype on another process. Most modern
22
     computers use two's complement for integers and IEEE format for floating point. Thus,
23
     communication using these size-specific datatypes will not entail loss of precision or trun-
24
     cation errors.
25
26
           Advice to users. Care is required when communicating in a heterogeneous environ-
27
           ment. Consider the following code:
28
29
           real(selected_real_kind(5)) x(100)
30
           size = storage_size(x) / 8
31
           call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
32
           if (myrank .eq. 0) then
33
                ... initialize x ...
34
               call MPI_SEND(x, xtype, 100, 1, ...)
35
           else if (myrank .eq. 1) then
36
                call MPI_RECV(x, xtype, 100, 0, ...)
37
           endif
38
39
           This may not work in a heterogeneous environment if the value of size is not the
40
           same on process 1 and process 0. There should be no problem in a homogeneous
41
           environment. To communicate in a heterogeneous environment, there are at least four
42
           options. The first is to declare variables of default type and use the MPI datatypes
43
           for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second
44
           is to use selected_real_kind or selected_int_kind and with the functions of the
45
           previous section. The third is to declare a variable that is known to be the same
46
           size on all architectures (e.g., selected_real_kind(12) on almost all compilers will
47
           result in an 8-byte representation). The fourth is to carefully check representation
48
           size before communication. This may require explicit conversion to a variable of size
```

that can be communicated and handshaking between sender and receiver to agree on a size.

Note finally that using the "external32" representation for I/O requires explicit attention to the representation sizes. Consider the following code:

```
real(selected_real_kind(5)) x(100)
size = storage_size(x) / 8
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',
                                                             &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,
                                                             &
                      MPI_INFO_NULL, fh, ierror)
                                                                             14
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',&
                           MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
                                                                             20
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
                                                                             21
                                                                             22
if (myrank .eq. 1) then
                                                                             23
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', MPI_MODE_RDONLY,
                                                                 X.
                 MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',&
                           MPI_INFO_NULL, ierror)
                                                                             27
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
                                                                             28
   call MPI_FILE_CLOSE(fh, ierror)
                                                                             29
endif
                                                                             30
```

If processes 0 and 1 are on different machines, this code may not work as expected if the size is different on the two machines. (End of advice to users.)

#### Problems With Fortran Bindings for MPI 19.1.10

This section discusses a number of problems that may arise when using MPI in a Fortran program. It is intended as advice to users, and clarifies how MPI interacts with Fortran. It is intended to clarify, not add to, this standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

1

 $\mathbf{2}$ 

3

4

56

7

8

9 10

11

12

13

15

16

17

18

19

 $^{24}$ 

25

26

31 32

33

34 35

36 37

38

39

40

41

42

43

44

45

46

47

- 1. An MPI subroutine with a choice argument may be called with different argument types. When using the mpi\_f08 module together with a compiler that supports Fortran 2008 + TS 29113, this problem is resolved.
- 2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of DIMENSION(...).
- 3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. This problem is solved with the use of the ASYNCHRONOUS attribute.
- 4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the ASYNCHRONOUS attribute as specified in TS 29113.
- 5. Several named "constants," such as MPI\_BOTTOM, MPI\_IN\_PLACE,
   MPI\_STATUS\_IGNORE, MPI\_STATUSES\_IGNORE, MPI\_ERRCODES\_IGNORE,
   MPI\_UNWEIGHTED, MPI\_WEIGHTS\_EMPTY, MPI\_ARGV\_NULL, and MPI\_ARGVS\_NULL
   are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 for more information.
  - 6. The memory allocation routine MPI\_ALLOC\_MEM cannot be used from Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, TYPE(C\_PTR) entities were added, which allow a standard-conforming implementation of the semantics of MPI\_ALLOC\_MEM. In MPI-3.0 and later, MPI\_ALLOC\_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The mpi\_f08 module only supports TYPE(C\_PTR) pointers.
    - Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.
      - MPI identifiers exceed 6 characters.
      - MPI identifiers may contain underscores after the first character.
      - MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
    - Many routines in MPI have KIND-parameterized integers (e.g., MPI\_ADDRESS\_KIND and MPI\_OFFSET\_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER\*8 or INTEGER should be used instead.
- <sup>44</sup> MPI-1 contained several routines that take address-sized information as input or return
   <sup>45</sup> address-sized information as output. In C such arguments were of type

MPI\_Aint and in Fortran of type INTEGER. On machines where integers are smaller than
 addresses, these routines can lose information. In MPI-2 the use of these functions has
 been deprecated and they have been replaced by routines taking INTEGER arguments of

 $41 \\ 42$ 

 $45 \\ 46$ 

KIND=MPI\_ADDRESS\_KIND. A number of MPI-2 functions also take INTEGER arguments of nondefault KIND. See Section 2.6 and Section 5.1.1 for more information.

Sections 19.1.11 through 19.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 19.1.7.

#### 19.1.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 19.1.6). In C, the use of void\* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 29113 (and later) together with the mpi\_f08 module, the problem is avoided by declaring choice arguments with TYPE(\*), DIMENSION(..), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
real x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning. When using either the mpi\_f08 or mpi module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi\_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI\_CART\_CREATE are declared as assumed size arrays INTEGER :: DIMS(\*) and LOGICAL :: PERIODS(\*).

```
USE mpi_f08 ! or USE mpi
INTEGER size
CALL MPI_Cart_create(comm_old, 1, size, .TRUE., .TRUE., comm_cart, ierror)
```

Although this is a nonconforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

19.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript triplets describe Fortran subarrays with or without strides, e.g.,

REAL a(100,100,100) CALL MPI\_Send(a(11:17, 12:99:3, 1:100), 7\*30\*100, MPI\_REAL, ...)

12	The handling of subscript triplets depends on the value of the constant MPI_SUBARRAYS_SUPPORTED:
$\frac{3}{4}$	• If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:
4 5 6 7	Choice buffer arguments are declared as TYPE(*), DIMENSION(). For example, consider the following code fragment:
8 9 10 11 12 13	REAL s(100), r(100) CALL MPI_Isend(s(1:100:5), 3, MPI_REAL,, rq, ierror) CALL MPI_Wait(rq, status, ierror) CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL,, rq, ierror) CALL MPI_Wait(rq, status, ierror)
14 15 16 17 18 19 20	In this case, the individual elements $s(1)$ , $s(6)$ , and $s(11)$ are sent between the start of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code will pass a descriptor to MPI_ISEND that allows MPI to operate directly on $s(1)$ , $s(6)$ , $s(11)$ ,, $s(96)$ . The called MPI_ISEND routine will take only the first three of these elements due to the type signature "3, MPI_REAL".
21 22 23 24 25 26 27 28 29 30 31	All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT, MPI_FILE_WRITE_ALL_BEGIN) behave as if the user-specified elements of choice buf- fers are copied to a contiguous scratch buffer in the MPI runtime environment. All datatype descriptions (in the example above, "3, MPI_REAL") read and store data from and to this virtual contiguous scratch buffer. Displacements in MPI derived datatypes are relative to the beginning of this virtual contiguous scratch buffer. Upon completion of a nonblocking receive operation (e.g., when MPI_WAIT on a correspond- ing MPI_Request returns), it is as if the received data has been copied from the virtual contiguous scratch buffer back to the noncontiguous application buffer. In the exam- ple above, r(1), r(6), and r(11) are guaranteed to be defined with the received data when MPI_WAIT returns.
32 33 34	Note that the above definition does not supercede restrictions about buffers used with nonblocking operations (e.g., those specified in Section 3.7.2).
35 36 37 38 39 40 41 42	Advice to implementors. The Fortran descriptor for TYPE(*), DIMENSION() arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of noncontiguous user buffers when the nonblocking operation is started, and release this buffer not before the nonblocking communication has completed (e.g., the MPI_WAIT routine). Efficient implementations may avoid such additional memory-to-memory data copying. (End of advice to implementors.)
42 43 44 45 46 47 48	<i>Rationale.</i> If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. ( <i>End of rationale.</i> )

• If MPI\_SUBARRAYS\_SUPPORTED equals .FALSE.:

In this case, the use of Fortran arrays with subscript triplets as actual choice buffer arguments in any nonblocking MPI operation (which also includes persistent request, and split collectives) may cause undefined behavior. They may, however, be used in blocking MPI operations.

Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

In Fortran, array data is not necessarily stored contiguously. For example, the array section A(1:N:2) involves only the elements of A with indices 1, 3, 5, .... The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., B(N)) or is of assumed size (e.g., B(\*)). If necessary, they do this by making a copy of the array into contiguous memory.<sup>1</sup>

Because MPI dummy buffer arguments are assumed-size arrays if MPI\_SUBARRAYS\_SUPPORTED equals .FALSE., this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

real a(100) call MPI\_IRECV(a(1:100:2), MPI\_REAL, 50, ...)

Since the first dummy argument to MPI\_IRECV is an assumed-size array (<type>buf(\*)), the array section a(1:100:2) is copied to a temporary before being passed to MPI\_IRECV, so that it is contiguous in memory. MPI\_IRECV returns immediately, and data is copied from the temporary back into the array a. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem for MPI\_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a "simply contiguous" section such as A(1:N) of such an array. ("Simply contiguous" is defined in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

According to the Fortran 2008 Standard, Section 6.5.4, a "simply contiguous" array section is

name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )

 $\mathbf{2}$ 

 $\overline{7}$ 

<sup>&</sup>lt;sup>1</sup>Technically, the Fortran standard is worded to allow noncontiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

	0	
1 2 3 4 5	S0 S1	That is, there are zero or more dimensions that are selected in full, then one dimension elected without a stride, then zero or more dimensions that are selected with a simple ubscript. The compiler can detect from analyzing the source code that the array is ontiguous. Examples are
6 7		A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)
8 9		Because of Fortran's column-major ordering, where the first index varies fastest, a simply contiguous" section of a contiguous array will also be contiguous.
10 11 12 13 14	so	The same problem can occur with a scalar argument. A compiler may make a copy of calar dummy arguments within a called procedure when passed as an actual argument of a choice buffer routine. That this can cause a problem is illustrated by the example
15 16 17 18 19 20		real :: a call user1(a,rq) call MPI_WAIT(rq,status,ierr) write (*,*) a
20 21 22 23 24		<pre>subroutine user1(buf,request) call MPI_IRECV(buf,,request,) end</pre>
25 26		f a is copied, MPI_IRECV will alter the copy when it completes the communication nd will not alter a itself.
27 28 29 30 31	e: se a	Note that copying will almost certainly occur for an argument that is a nontrivial expression (one with at least one operator or function call), a section that does not elect a contiguous part of its parent (e.g., $A(1:n:2)$ ), a pointer whose target is such section, or an assumed-shape array that is (directly or indirectly) associated with uch a section.
32 33		f a compiler option exists that inhibits copying of arguments, in either the calling or alled procedure, this must be employed.
34 35 36 37 38 39 40 41	sl sc N is	f a compiler makes copies in the calling procedure of arguments that are explicit- hape or assumed-size arrays, "simply contiguous" array sections of such arrays, or calars, and if no compiler option exists to inhibit such copying, then the compiler annot be used for applications that use MPI_GET_ADDRESS, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications hat use memory references across subroutine calls as in the example above.
42 43	19.1.13	8 Problems Due to Data Copying and Sequence Association with Vector Subscripts
44 45 46		n arrays with <b>vector</b> subscripts describe subarrays containing a possibly irregular elements
47 48		L a(100) L MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL,)

CHAPTER 19. LANGUAGE BINDINGS

Fortran arrays with a vector subscript must not be used as actual choice buffer arguments in any nonblocking or split collective MPI operations. They may, however, be used in blocking MPI operations.

#### 19.1.14 Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI\_BOTTOM. The complete list can be found in Section 2.5.4. In C, these are implemented as constant pointers, usually as NULL and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran, using special values for the constants (e.g., by defining them through **parameter** statements) is not possible because an implementation cannot distinguish these values from valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran INTENT: with USE mpi\_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an INTENT is not specified.

#### 19.1.15 Fortran Derived Types

MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.

The following code fragment shows some possible ways to send scalars or arrays of interoperable derived types in Fortran. The example assumes that all data is passed by address.

```
type, BIND(C) :: mytype
   integer :: i
  real :: x
  double precision :: d
   logical :: 1
end type mytype
type(mytype) :: foo, fooarr(5)
integer :: blocklen(4), type(4)
integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent
call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%1, disp(4), ierr)
base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
```

 $\mathbf{2}$ 

```
1
     disp(3) = disp(3) - base
\mathbf{2}
     disp(4) = disp(4) - base
3
4
     blocklen(1) = 1
\mathbf{5}
     blocklen(2) = 1
6
     blocklen(3) = 1
7
     blocklen(4) = 1
8
9
     type(1) = MPI_INTEGER
10
     type(2) = MPI_REAL
11
     type(3) = MPI_DOUBLE_PRECISION
12
     type(4) = MPI_LOGICAL
13
14
     call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
15
     call MPI_TYPE_COMMIT(newtype, ierr)
16
17
     call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
18
     ! or
19
     call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
20
     ! expects that base == address(foo%i) == address(foo)
21
22
     call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
23
     call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
^{24}
     extent = disp(2) - disp(1)
25
     1b = 0
26
     call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
27
     call MPI_TYPE_COMMIT(newarrtype, ierr)
28
29
     call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
30
```

Using the derived type variable foo instead of its first basic type element foo%i may be impossible if the MPI library implements choice buffer arguments through overloading instead of using TYPE(\*), DIMENSION(..), or through a nonstandardized extension such as !\$PRAGMA IGNORE\_TKR; see Section 19.1.6.

To use a derived type in an array requires a correct extent of the datatype handle 35 to take care of the alignment rules applied by the compiler. These alignment rules may 36 imply that there are gaps between the components of a derived type, and also between the 37 subsuguent elements of an array of a derived type. The extent of an interoperable derived 38 type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may 39 be different because C and Fortran may apply different alignment rules. As recommended 40 in the advice to users in Section 5.1.6, one should add an additional fifth structure element 41 with one numerical storage unit at the end of this structure to force in most cases that 42the array of structures is contiguous. Even with such an additional element, one should 43 keep this resizing due to the special alignment rules that can be used by the compiler for 44 structures, as also mentioned in this advice. 45

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER type components may exist. In this case, the **base** address in the example must be changed to become the address of **foo** instead of **foo%i**, because the Fortran compiler may rearrange type components or add padding. Sending the structure **foo** should then also be performed by providing it (and not **foo%i**) as actual argument for MPI\_Send.

#### 19.1.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI\_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the mpi\_f08 module, the mpi module, and the mpif.h include file.

This section shows four problematic usage areas (the abbreviations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests (Nonbl.).
- Use of one-sided routines (1-sided).
- Use of MPI parallel file I/O split collective operations (Split).
- Use of MPI\_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes (*Bottom*).

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 19.1.17.
- Temporary data movement and temporary memory modifications; see Section 19.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 19.1.19.

Table 19.2 shows the only usage areas where these optimization problems may occur.

Optimization	n	may cause a problem in		
	fc	ollowing u	sage are	eas
	Nonbl.	1-sided	Split	Bottom
Code movement	yes	yes	no	yes
and register optimization				
Temporary data movement	yes	yes	yes	no
Permanent data movement	yes	yes	yes	yes

Table 19.2: Occurrence of Fortran optimization problems in several usage areas The solutions in the following sections are based on compromises: 1

 $\mathbf{2}$ 

3

4

5 6

7

8

9

10

11

12

13

14

15

16

17 18

19

20 21

22 23

24

25 26

27 28

29 30

 $^{31}$ 

32 33

 $45 \\ 46$ 

• to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on pages 827–832,

- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 19.1.7.

## 19.1.17 Problems with Code Movement and Register Optimization

Nonblocking Operations

If a variable is local to a Fortran subroutine (i.e., not in a module or a COMMON block), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register which held a valid copy of such a variable before the call will still hold a valid copy on return.

```
Example 19.1 Fortran 90 register optimization—extreme.
Source
                           compiled as
                                                       or compiled as
REAL :: buf, b1
                           REAL :: buf, b1
                                                       REAL :: buf, b1
call MPI_IRECV(buf,..req)
                           call MPI_IRECV(buf,..req)
                                                       call MPI_IRECV(buf,..req)
                           register = buf
                                                       b1 = buf
                                                       call MPI_WAIT(req,..)
call MPI_WAIT(req,..)
                            call MPI_WAIT(req,..)
b1 = buf
                           b1 = register
```

25 26

Example 19.1 shows extreme, but allowed, possibilities. MPI\_WAIT on a concurrent thread modifies buf between the invocation of MPI\_IRECV and the completion of MPI\_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI\_IRECV has returned, and may schedule the load of buf earlier than typed in the source. The compiler has no reason to avoid using a register to hold buf across the call to MPI\_WAIT. It also may reorder the instructions as illustrated in the rightmost column.

```
Example 19.2 Similar example with MPI_ISEND
```

```
Source
                            compiled as
                                                        with a possible MPI-internal
                                                        execution sequence
REAL :: buf, copy
                            REAL :: buf, copy
                                                        REAL :: buf, copy
buf = val
                            buf = val
                                                        buf = val
call MPI_ISEND(buf,..req)
                            call MPI_ISEND(buf,..req)
                                                        addr = &buf
copy = buf
                            copy= buf
                                                        copy = buf
                            buf = val_overwrite
                                                        buf = val_overwrite
call MPI_WAIT(req,..)
                            call MPI_WAIT(req,..)
                                                        call send(*addr) ! within
                                                                          ! MPI_WAIT
buf = val_overwrite
```

44 45 46

47

48

Due to valid compiler code movement optimizations in Example 19.2, the content of buf may already have been overwritten by the compiler when the content of buf is sent.

1

 $\mathbf{2}$ 

3

4

5 6

7 8

9 10

11

12

13

14

15

16 17

18 19

20

21

22

23

 $^{24}$ 

27

28

29

30

 $^{31}$ 

32

33 34

35

36

37 38

39

40

41

42

The code movement is permitted because the compiler cannot detect a possible access to buf in MPI\_WAIT (or in a second thread between the start of MPI\_ISEND and the end of MPI\_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI\_WAIT to before MPI\_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the MPI\_XXX\_BEGIN and MPI\_XXX\_END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI\_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

#### Persistent Operations

With persistent requests, the buffer argument is hidden from the MPI\_START and MPI\_STARTALL calls, i.e., the Fortran compiler may move buffer accesses across the MPI\_START or MPI\_STARTALL call, similar to the MPI\_WAIT call as described in the Nonblocking Operations subsection in Section 19.1.17.

#### One-sided Communication

An example with instruction reordering due to register optimization can be found in Section 12.7.4.

#### MPI\_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI\_SEND, MPI\_RECV, etc., that hides the actual variables involved in the communication. MPI\_BOTTOM with an MPI\_Datatype containing *absolute addresses* is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI\_GET\_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 19.3 shows what Fortran compilers are allowed to do.

In Example 19.3, the compiler does not invalidate the register because it cannot see that MPI\_RECV changes the value of buf. The access to buf is hidden by the use of MPI\_GET\_ADDRESS and MPI\_BOTTOM.

In Example 19.4, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI\_SEND statement is interfering because the load access to buf is hidden by the usage of MPI\_BOTTOM.

#### Solutions

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations

 $\mathbf{2}$ 

 $\mathbf{5}$ 

 $^{31}$ 

**Example 19.3** Fortran 90 register optimization. This source ... can be compiled as: call MPI\_GET\_ADDRESS(buf, bufaddr, call MPI\_GET\_ADDRESS(buf,...) ierror) call MPI\_TYPE\_CREATE\_STRUCT(1,1, call MPI\_TYPE\_CREATE\_STRUCT(...) bufaddr, MPI\_REAL,type,ierror) call MPI\_TYPE\_COMMIT(type,ierror) call MPI\_TYPE\_COMMIT(...) val\_old = buf register = buf val\_old = register call MPI\_RECV(MPI\_BOTTOM,1,type,...) call MPI\_RECV(MPI\_BOTTOM,...) val\_new = buf val\_new = register

<b>Example 19.4</b> Similar example with MPI_S	END
This source	can be compiled as:
! buf contains val_old buf = val_new	! buf contains val_old
<pre>call MPI_SEND(MPI_BOTTOM,1,type,)</pre>	<pre>call MPI_SEND()</pre>
! with buf as a displacement in type	! i.e. val_old is sent
	!
	! buf=val_new is moved to here
	! and detected as dead code
	! and therefore removed
	!
buf = val_overwrite	<pre>buf = val_overwrite</pre>

with the send and receive buffers used in nonblocking operations, or in operations in which MPI\_BOTTOM is used, or if datatype handles that combine several variables are used:

- Use of the Fortran ASYNCHRONOUS attribute.
- Use of the helper routine MPI\_F\_SYNC\_REG, or an equivalent user-written dummy routine.
- Declare the buffer as a Fortran module variable or within a Fortran common block.
- Use of the Fortran **VOLATILE** attribute.

Each of these methods solves the problems of code movement and register optimization, but may incur various degrees of performance impact, and may not be usable in every application context. These methods may not be guaranteed by the Fortran standard, but they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated compiler suite according to the requirements listed in Section 19.1.7. The performance impact of using MPI\_F\_SYNC\_REG is expected to be low, that of using module variables

828

1

2 3

4

5

6

7

8

9

10

11

12

13

33

34 35

36

37

38 39

40 41

or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the VOLATILE attribute is expected to be high or very high. Note that there is one attribute that cannot be used for this purpose: the Fortran TARGET attribute does not solve code movement problems in MPI applications.

#### The Fortran ASYNCHRONOUS Attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with ASYNCHRONOUS (which is mandatory for the mpi\_f08 module, with allowable exceptions listed in Section 19.1.6), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI\_ASYNC\_PROTECTS\_NONBLOCKING is set to .TRUE. if both the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 29113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to .FALSE ..

The ASYNCHRONOUS attribute has some restrictions. Section 5.4.2 of the TS 29113 specifies:

"Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent.

Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed."

In Example 19.5 Case (a) on page 835, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable b is the pending communication affector and is usable for input and output asynchronous communication

1 2

3

4

5 6

 $\overline{7}$ 

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26 27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41

42 43

44

45

46

47

1 between the MPI\_IXXX routines and MPI\_Waitall. Case (a) works fine because the read  $\mathbf{2}$ accesses to **b** occur after the communication has completed.

3 In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to 4 a pending communication affector while input communication (i.e., the two MPI\_Irecv  $\mathbf{5}$ calls) is pending. This is a contradiction to the rule that for input communication, a 6 pending communication affector shall not be referenced. The problem can be solved by using 7separate variables for the halos and the inner array, or by splitting a common array into 8 disjoint subarrays which are passed through different dummy arguments into a subroutine, 9 as shown in Example 19.9.

10 If one does not overlap communication and computation on the same variable, then all 11optimization problems can be solved through the ASYNCHRONOUS attribute.

12The problems with MPI\_BOTTOM, as shown in Example 19.3 and Example 19.4, can 13also be solved by declaring the buffer **buf** with the **ASYNCHRONOUS** attribute.

In some MPI routines, a buffer dummy argument is defined as ASYNCHRONOUS to guarantee passing by reference, provided that the actual argument is also defined as ASYNCHRONOUS.

17Calling MPI\_F\_SYNC\_REG

The compiler may be prevented from moving a reference to a buffer across a call to an 19MPI subroutine by surrounding the call by calls to an external subroutine with the buffer 20as an actual argument. The MPI library provides the MPI\_F\_SYNC\_REG routine for this 21purpose; see Section 19.1.8. 22

2324

14

15

16

18

2526

> 36

37

38

39 40

41

42

43

• The problems illustrated by the Examples 19.1 and 19.2 can be solved by calling MPI\_F\_SYNC\_REG(buf) once immediately after MPI\_WAIT.

b	Example 19.1	Example 19.2
7	can be solved with	can be solved with
8	<pre>call MPI_IRECV(buf,req)</pre>	buf = val
9		call MPI_ISEND(buf,req)
1		copy = buf
2	<pre>call MPI_WAIT(req,)</pre>	<pre>call MPI_WAIT(req,)</pre>
-	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
4	b1 = buf	<pre>buf = val_overwrite</pre>

The call to MPI\_F\_SYNC\_REG(buf) prevents moving the last line before the MPI\_WAIT call. Further calls to MPI\_F\_SYNC\_REG(buf) are not needed because it is still correct if the additional read access copy=buf is moved below MPI\_WAIT and before buf=val\_overwrite.

• The problems illustrated by the Examples 19.3 and 19.4 can be solved with two additional MPI\_F\_SYNC\_REG(buf) statements; one directly before MPI\_RECV/ MPI\_SEND, and one directly after this communication operation.

44	Example 19.3	Example 19.4
45	can be solved with	can be solved with
46	call MPI_F_SYNC_REG(buf)	call MPI_F_SYNC_REG(buf)
47	<pre>call MPI_RECV(MPI_BOTTOM,)</pre>	<pre>call MPI_SEND(MPI_BOTTOM,)</pre>
48	<pre>call MPI_F_SYNC_REG(buf)</pre>	call MPI_F_SYNC_REG(buf)

The first call to MPI\_F\_SYNC\_REG(buf) is needed to finish all load and store references to buf prior to MPI\_RECV/MPI\_SEND; the second call is needed to assure that any subsequent access to buf is not moved before MPI\_RECV/MPI\_SEND.

• In the example in Section 12.7.4, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 19.1, i.e., a call to MPI\_F\_SYNC\_REG(bbbb) is needed after the second MPI\_WIN\_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI\_WIN\_FENCE. In Process 2, both calls to MPI\_WIN\_FENCE together act as a communication call with MPI\_BOTTOM as the buffer. That is, before the first fence and after the second fence, a call to MPI\_F\_SYNC\_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI\_WIN\_FENCE. Using MPI\_GET instead of MPI\_PUT, the same calls to MPI\_F\_SYNC\_REG are necessary.

Source of Process 1	Source of Process 2	15
bbbb = 777	buff = 999	16
	call MPI_F_SYNC_REG(buff)	17
call MPI_WIN_FENCE	call MPI_WIN_FENCE	18
call MPI_PUT(bbbb		19
into buff of process 2)		20
		21
call MPI_WIN_FENCE	call MPI_WIN_FENCE	22
call MPI_F_SYNC_REG(bbbb)	call MPI_F_SYNC_REG(buff)	23
	ccc = buff	24
		~ ~

- The temporary memory modification problem, i.e., Example 19.6, can **not** be solved with this method.
- A User Defined Routine Instead of MPI\_F\_SYNC\_REG

Instead of MPI\_F\_SYNC\_REG, one can also use a user defined external subroutine, which is separately compiled:

> subroutine DD(buf) integer buf end

Note that if the INTENT is declared in an explicit interface for the external subroutine, it must be OUT or INOUT. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, a call to MPI\_RECV with MPI\_BOTTOM as buffer might be replaced by

```
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI\_F\_SYNC\_REG or one of the other possibilities. In an existing application, calls to

1 such a user-written routine should be substituted by a call to MPI\_F\_SYNC\_REG because  $\mathbf{2}$ the user-written routine may not be implemented in accordance with the rules specified in 3

- Section 19.1.7.
- 4 5

6

13

14

15

1617

18

## Module Variables and COMMON Blocks

An alternative to the previously mentioned methods is to put the buffer or variable into a 7 module or a common block and access it through a USE or COMMON statement in each scope 8 where it is referenced, defined or appears as an actual argument in a call to an MPI routine. 9 The compiler will then have to assume that the MPI procedure may alter the buffer or 10 variable, provided that the compiler cannot infer that the MPI procedure does not reference 11 the module or common block. 12

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI\_BOTTOM and derived datatype handles.
- Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.
- 192021

22

## The (Poorly Performing) Fortran VOLATILE Attribute

The VOLATILE attribute gives the buffer or variable the properties needed to avoid register 23optimization or code movement problems, but it may inhibit optimization of any code 24containing references or definitions of the buffer or variable. On many modern systems, the 25performance impact will be large because not only register, but also cache optimizations 26will not be applied. Therefore, use of the VOLATILE attribute to enforce correct execution 27of MPI programs is discouraged. 28

29 30

## The Fortran TARGET Attribute

 $^{31}$ The TARGET attribute does not solve the code movement problem because it is not specified 32 for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that 33 the application program specifies the TARGET attribute for an actual buffer argument used 34in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer 35 reference to this buffer exists. 36

Rationale. The Fortran standardization body decided to extend the ASYNCHRONOUS attribute within the TS 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the TARGET attribute. (End of rationale.)

39 40

37

38

 $^{41}$ 42

#### Temporary Data Movement and Temporary Memory Modification 19.1.18

The compiler is allowed to temporarily modify data in memory. Normally, this problem 43 may occur only when overlapping communication and computation, as in Example 19.5, 44 Case (b) on page 835. Example 19.6 also shows a possibility that could be problematic. 45

In the compiler-generated, possible optimization in Example 19.7, buf(100,100) from 46 Example 19.6 is equivalenced with the 1-dimensional array buf\_1dim(10000). The nonblock-47ing receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused 48

loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 19.8 shows a second possible optimization. The whole array is temporarily moved to local\_buf.

When storing local\_buf back to the original location buf, then this implies overwriting the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this storing back of local\_buf is therefore likely to interfere with asynchronously received data in buf(1,1:100).

Note that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 12.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations between the MPI\_XXX\_BEGIN and MPI\_XXX\_END calls; see Section 14.4.5.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 829 of Section 19.1.17, the ASYNCHRONOUS attribute can prevent compiler optimization with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 19.9 and in Example 19.10.

Note also that the methods

- calling MPI\_F\_SYNC\_REG (or such a user-defined routine),
- using module variables and COMMON blocks, and
- the TARGET attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the code fragments shown in Example 19.6 and 19.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring **buf** as **VOLATILE** because the **VOLATILE** implies that all accesses to any storage unit (word) of **buf** must be directly done in the main memory exactly in the sequence defined by the application program. The **VOLATILE** attribute prevents all register and cache optimizations. Therefore, **VOLATILE** may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the ASYNCHRONOUS attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

Rationale. This is a strong restriction for application programs. To weaken this <sup>46</sup> restriction, a new or modified asynchronous feature in the Fortran language would <sup>47</sup> be necessary: an asynchronous attribute that can be used on parts of an array and <sup>48</sup>

1

 $\mathbf{2}$ 

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8 9

10

11

12

13

14 15

16

17 18

19 20

21

22

23

 $^{24}$ 

25 26

27

28 29

30 31

32

33

34

35

36

37

38

39

40

41

42

43

44

together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 19.9 (which is a solution for the problem shown in Example 19.5 and 5in Example 19.10 (which is a solution for the problem shown in Example 19.8), the ar-6 ray is split into inner and halo part and both disjoint parts are passed to a subroutine 7 separated\_sections. This routine overlaps the receiving of the halo data and the calcu-8 lations on the inner part of the array. In a second step, the whole array is used to do the 9 calculation on the elements where inner+halo is needed. Note that the halo and the inner 10 area are strided arrays. Those can be used in nonblocking communication only with a TS 11 29113 based MPI library. 12

 $13 \\ 14$ 

15

16

17

18

19 20

21

22

23

24 25

26

27

28

1

2

3

4

## 19.1.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. An implementation with automatic garbage collection is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with MPI\_BOTTOM.
- All nonblocking MPI operations if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the ASYNCHRONOUS attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 19.1.7.

29 30 31

## 19.1.20 Comparison with C

32 In C, subroutines which modify variables that are not in the argument list will not cause 33 register optimization problems. This is because taking pointers to storage objects by using 34the & operator and later referencing the objects by indirection on the pointer is an integral 35 part of the language. A C compiler understands the implications, so that the problem should 36 not occur, in general. However, some compilers do offer optional aggressive optimization 37 levels which may not be safe. Problems due to temporary memory modifications can also 38occur in C. As above, the best advice is to avoid the problem: use different variables for 39 buffers in nonblocking MPI operations and computation that is executed while a nonblocking 40operation is pending.

- 41
- 42
- 43
- $44 \\ 45$
- 46
- 46
- 47 48

```
Example 19.5 Protecting nonblocking communication with the ASYNCHRONOUS attribute.
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)
                               ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b( 1), ..., left, ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)
#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
  CALL MPI_Waitall(4, req, ...)
  DO i=1,100 ! compute all new local data
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
#endif
#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
  DO i=2,99 ! compute only elements for which halo data is not needed
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
  CALL MPI_Waitall(4, req, ...)
  i=1 ! compute leftmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
  i=100 ! compute rightmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
#endif
```

1

2 3

4

5

6

7

8

9

10

11

12

13 14 15

16

17

18

19

20

21

22 23

24

25

26

27 28

29

30

 $^{31}$ 

32

33

34

```
1
     Example 19.6 Overlapping Communication and Computation.
\mathbf{2}
3
     USE mpi_f08
4
     REAL :: buf(100,100)
\mathbf{5}
6
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
7
     DO j=1,100
8
       DO i=2,100
9
          buf(i,j)=...
10
        END DO
11
     END DO
12
13
     CALL MPI_Wait(req,...)
14
15
16
     Example 19.7 The compiler may substitute the nested loops through loop fusion.
17
18
     REAL :: buf(100,100), buf_1dim(10000)
19
     EQUIVALENCE (buf(1,1), buf_1dim(1))
20
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
21
     tmp(1:100) = buf(1,1:100)
^{22}
     DO j=1,10000
23
^{24}
       buf_1dim(h)=...
25
     END DO
26
     buf(1,1:100) = tmp(1:100)
27
28
     CALL MPI_Wait(req,...)
29
30
31
     Example 19.8 Another optimization is based on the usage of a separate memory storage
32
     area, e.g., in a GPU.
33
34
     REAL :: buf(100,100), local_buf(100,100)
35
36
     CALL MPI_Irecv(buf(1,1:100),..., req,...)
37
     local_buf = buf
38
     DO j=1,100
39
       DO i=2,100
40
          local_buf(i,j)=...
41
       END DO
42
43
     END DO
44
     buf = local_buf ! may overwrite asynchronously received
45
                        ! data in buf(1,1:100)
46
     CALL MPI_Wait(req,...)
47
```

```
Example 19.9 Using separated variables for overlapping communication and computation
to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.
USE mpi_f08
REAL :: b(0:101)
                     ! elements 0 and 101 are halo cells
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
INTEGER :: i
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
i=1 ! compute leftmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
i=100 ! compute rightmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
END
SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
USE mpi_f08
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
REAL :: bnew(0:101) ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left, right,...)
CALL MPI_Irecv(b_lefthalo ( 0), ..., left, ..., req(1), ...)
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
! b_lefthalo and b_righthalo is written asynchronously.
! There is no other concurrent access to b_lefthalo and b_righthalo.
                             ..., left, ..., req(3), ...)
CALL MPI_Isend(b_inner( 1),
CALL MPI_Isend(b_inner(100),
                                ..., right, ..., req(4), ...)
DO i=2,99 ! compute only elements for which halo data is not needed
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
  ! b_inner is read and sent at the same time.
  ! This is allowed based on the rules for ASYNCHRONOUS.
```

END SUBROUTINE

CALL MPI\_Waitall(4, req,...)

END DO

1

2

3 4

5

6

7

8

9

10

11

12

13

14 15

16

17

18

19

20

21

22

23 24

25

26

27 28

29 30

31

32

33

34

35 36

```
1
     Example 19.10 Protecting GPU optimizations with the ASYNCHRONOUS attribute.
\mathbf{2}
3
     USE mpi_f08
4
     REAL :: buf(100,100)
\mathbf{5}
     CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
6
      END
\overline{7}
8
     SUBROUTINE separated_sections(buf_halo, buf_inner)
9
     REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
10
     REAL :: buf_inner(2:100,1:100)
11
     REAL :: local_buf(2:100,100)
12
13
     CALL MPI_Irecv(buf_halo(1,1:100),..., req,...)
14
     local_buf = buf_inner
15
     DO j=1,100
16
        DO i=2,100
17
          local_buf(i,j)=...
18
19
        END DO
20
     END DO
21
     buf_inner = local_buf ! buf_halo is not touched!!!
^{22}
23
     CALL MPI_Wait(req,...)
^{24}
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
```

5

6

7 8

9

10 11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

42

43

19.2	Support for Large Count and Large Byte Displacement in MPI Lan- guage Bindings
	lowing types, which were used prior to MPI-4.0, have been deemed too small to hold that applications wish to use:
• T	`he C int type and the Fortran INTEGER type were used for <i>count</i> parameters.
	The C int type and the Fortran INTEGER type were used for some parameters that epresent <i>byte displacement</i> in memory.
u	The C MPI_Aint type and the Fortran INTEGER(KIND=MPI_ADDRESS_KIND) type were sed for some parameters that represent <i>byte displacement</i> in files (e.g., in constructors f MPI datatypes that can be used with files).
types v polymo large co support Fo MPI_T MPI_T	er to avoid breaking backwards compatibility, this version of MPI supports larger via separate additional MPI procedures in C (suffixed with "_c") and via interface orphism in Fortran when using USE mpi_f08. For better readability, all Fortran ount procedure declarations are marked with a comment "!(_c)". No polymorphic t for larger types is provided in Fortran when using mpif.h and use mpi. r the large count versions of three datatype constructors, YPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, and YPE_CREATE_STRUCT, absolute addresses shall not be used to specify byte dis- ents since the parameter is of type MPI_COUNT instead of type MPI_AINT (see Sec- 5.8).
In MPI_T separat in Fort	addition, the functions MPI_TYPE_GET_ENVELOPE and YPE_GET_CONTENTS also support large count types via <i>additional parameters</i> in the additional MPI procedures in C (suffixed with "_c") and interface polymorphism ran when using USE mpi_f08 (see Section 5.1.13).

Further, the callbacks of type MPI\_User\_function and MPI\_Datarep\_conversion\_function 29 also support large count types via separate additional callback prototypes in C (suffixed 30 with "\_c") and multiple abstract interfaces in Fortran when using USE mpi\_f08 (see Sec- $^{31}$ tions 6.9.5 and 14.5.3, respectively). An additional large count predefined callback function 32 MPI\_CONVERSION\_FN\_NULL\_C is provided within each of these two language bindings. 33

In C bindings, for each MPI procedure that had at least one *count* or *byte displacement* 34 parameter that used the int and/or MPI\_Aint types prior to MPI-4.0, an additional MPI 35procedure is provided, with the same name but suffixed by "\_c". The MPI procedure 36 without the "\_c" token has the same name and parameter types as versions prior to MPI-37 4.0. The "\_c" suffixed MPI procedure has MPI\_Count for all count parameters, MPI\_Aint 38 for parameters that represent byte displacement in memory, MPI\_Offset for parameters 39 that represent byte displacement in files, and MPI\_Count for parameters that may represent 40 byte displacement in both memory and files. 41

In Fortran, when using USE mpi\_f08, for each MPI procedure that had at least one count or byte displacement parameter that used the INTEGER or

INTEGER(KIND=MPI\_ADDRESS\_KIND) types prior to MPI-4.0, a polymorphic interface con-44taining two specific procedures is provided. One of the specific procedures has the same 45name and dummy parameter types as in versions prior to MPI-4.0. INTEGER and/or 46INTEGER (KIND=MPI\_ADDRESS\_KIND) for count and byte displacement parameters. The other 47specific procedure has the same name followed by "\_c", and then suffixed by the token 48

1 specified in Table 19.1 for USE mpi\_f08. It also has INTEGER(KIND=MPI\_COUNT\_KIND)  $\mathbf{2}$ for all *count* parameters, INTEGER(KIND=MPI\_ADDRESS\_KIND) for parameters that repre-3 sent byte displacement in memory, INTEGER(KIND=MPI\_OFFSET\_KIND) for parameters that 4 represent byte displacement in files, and INTEGER(KIND=MPI\_COUNT\_KIND) for parame- $\mathbf{5}$ ters that may represent byte displacement in both memory and files (for more details 6 on specific Fortran procedure names and related calling conventions, refer to Table 19.1  $\overline{7}$ in Section 19.1.5). There is one exception: if the type signatures of the two specific 8 procedures are identical (e.g., if INTEGER(KIND=MPI\_COUNT\_KIND) is the same type as 9 INTEGER(KIND=MPI\_ADDRESS\_KIND), then the implementation shall not provide the "\_c" 10 specific procedure.

<sup>11</sup> It is erroneous to directly invoke the "\_c" specific procedures in the Fortran mpi\_f08 <sup>12</sup> module with the exception of the following procedures: MPI\_Op\_create\_c and

<sup>13</sup> MPI\_Register\_datarep\_c.

<sup>14</sup> In older Fortran bindings (mpif.h and use mpi), no new interfaces and no new specific <sup>15</sup> procedures for larger types are provided beyond what existed in MPI-3.1; all MPI procedures <sup>16</sup> have the same types as in the versions prior to MPI-4.0.

17 18 19

20

21

## 19.3 Language Interoperability

19.3.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication
 used between a parallel client and a parallel server. It should be possible to code the server
 in one language and the clients in another language. To do so, communications should be
 possible between applications written in different languages.

31

38

41

42

43

There are several issues that need to be addressed in order to achieve interoperability.

 $_{33}^{_{32}}$  Initialization We need to specify how the MPI environment is initialized for all languages.

- Interlanguage passing of MPI opaque objects We need to specify how MPI object
   handles are passed between languages. We also need to specify what happens when
   an MPI object is accessed in one language, to retrieve information (e.g., attributes)
   set in another language.
- Interlanguage communication We need to specify how messages sent in one language
   can be received in another language.

It is highly desirable that the solution for interlanguage interoperability be extensible to new languages, should MPI bindings be defined for such languages.

<sup>44</sup><sub>45</sub> 19.3.2 Assumptions

We assume that conventions exist for programs written in one language to call routines
 written in another language. These conventions specify how to link routines in different
 languages into one program, how to call functions in a different language, how to pass

arguments between languages, and the correspondence between basic datatypes in different languages. In general, these conventions will be implementation dependent. Furthermore, not every basic datatype may have a matching type in other languages. For example, C character strings may not be compatible with Fortran CHARACTER variables. However, we assume that a Fortran INTEGER, as well as a (sequence associated) Fortran array of INTEGERS, can be passed to a C program. We also assume that Fortran and C have addresssized integers. This does not mean that the default-size integers are the same size as default-sized pointers, but only that there is some way to hold (and pass) a C address in a Fortran integer. It is also assumed that INTEGER(KIND=MPI\_OFFSET\_KIND) can be passed from Fortran to C as MPI\_Offset.

#### 19.3.3 Initialization

A call to MPI\_INIT or MPI\_INIT\_THREAD, from any language, initializes MPI for execution in all languages.

Advice to users. Certain implementations use the (inout) argc, argv arguments of the C version of MPI\_INIT in order to propagate values for argc and argv to all executing processes. Use of the Fortran version of MPI\_INIT to initialize MPI may result in a loss of this ability. (End of advice to users.)

The function MPI\_INITIALIZED returns the same answer in all languages.

The function MPI\_FINALIZE finalizes the MPI environments for all languages.

The function MPI\_FINALIZED returns the same answer in all languages.

The function MPI\_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI\_INIT. E.g., MPI\_COMM\_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

Advice to users. The use of several languages in one MPI program may require the use of special options at compile and/or link time. (End of advice to users.)

Advice to implementors. Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code needs to perform initialization for a language only if that language library is loaded. (End of advice to *implementors.*)

#### 19.3.4 Transfer of Handles

Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C handles in Fortran.

The type definition MPI\_Fint is provided in C for an integer of the size that matches a 44Fortran INTEGER; usually, MPI\_Fint will be equivalent to int. With the Fortran mpi module or the mpif.h include file, a Fortran handle is a Fortran INTEGER value that can be used in the following conversion functions. With the Fortran mpi\_f08 module, a Fortran handle is a

1

 $\mathbf{2}$ 

3

4

5

6  $\overline{7}$ 

8

9

10

11 12

13

14

1516

17

18

19

2021

22

23

 $^{24}$ 

25

26

27

28

29

30 31

32

33 34

35

36

37

38

39 40

41

42

43

45

46

1BIND(C) derived type that contains an INTEGER component named MPI\_VAL. This INTEGER  $\mathbf{2}$ value can be used in the following conversion functions. 3 The following functions are provided in C to convert from a Fortran communicator 4 handle (which is an integer) to a C communicator handle, and vice versa. See also Sec- $\mathbf{5}$ tion 2.6.4. 6 C binding 7MPI\_Comm MPI\_Comm\_f2c(MPI\_Fint comm) 8 If comm is a valid Fortran handle to a communicator, then MPI\_Comm\_f2c returns a 9 valid C handle to that same communicator; if  $comm = MPI_COMM_NULL$  (Fortran value), 10 then MPI\_Comm\_f2c returns a null C handle; if comm is an invalid Fortran handle, then 11 MPI\_Comm\_f2c returns an invalid C handle. 12MPI\_Fint MPI\_Comm\_c2f(MPI\_Comm comm) 1314The function MPI\_Comm\_c2f translates a C communicator handle into a Fortran handle 15to the same communicator; it maps a null handle into a null handle and an invalid handle 16into an invalid handle. 17Similar functions are provided for the other types of opaque objects. 18MPI\_Datatype MPI\_Type\_f2c(MPI\_Fint datatype) 19MPI\_Fint MPI\_Type\_c2f(MPI\_Datatype datatype) 2021MPI\_Group MPI\_Group\_f2c(MPI\_Fint group) 22MPI\_Fint MPI\_Group\_c2f(MPI\_Group group) 23 $^{24}$ MPI\_Request MPI\_Request\_f2c(MPI\_Fint request) 25MPI\_Fint MPI\_Request\_c2f(MPI\_Request request) 2627MPI\_File MPI\_File\_f2c(MPI\_Fint file) 28MPI\_Fint MPI\_File\_c2f(MPI\_File file)  $^{29}$ 30 MPI\_Win MPI\_Win\_f2c(MPI\_Fint win)  $^{31}$ 32MPI\_Fint MPI\_Win\_c2f(MPI\_Win win) 33 MPI\_Op MPI\_Op\_f2c(MPI\_Fint op) 34MPI\_Fint MPI\_Op\_c2f(MPI\_Op op) 3536 MPI\_Info MPI\_Info\_f2c(MPI\_Fint info) 37 38MPI\_Fint MPI\_Info\_c2f(MPI\_Info info) 39 MPI\_Errhandler MPI\_Errhandler\_f2c(MPI\_Fint errhandler) 4041MPI\_Fint MPI\_Errhandler\_c2f(MPI\_Errhandler errhandler) 42MPI\_Message MPI\_Message\_f2c(MPI\_Fint message) 4344MPI\_Fint MPI\_Message\_c2f(MPI\_Message message) 45MPI\_Session MPI\_Session\_f2c(MPI\_Fint session) 4647MPI\_Fint MPI\_Session\_c2f(MPI\_Session session) 48

**Example 19.11** The example below illustrates how the Fortran MPI function MPI\_TYPE\_COMMIT can be implemented by wrapping the C MPI function MPI\_Type\_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)
INTEGER :: DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END
/* C wrapper */
void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)
{
    MPI_Datatype datatype;
    datatype = MPI_Type_f2c(*f_handle);
    *ierr = (MPI_Fint)MPI_Type_commit(&datatype);
    *f_handle = MPI_Type_c2f(datatype);
    return;
}
```

The same approach can be used for all other MPI functions. The call to MPI\_XXX\_f2c (resp. MPI\_XXX\_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

*Rationale.* The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.

Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

## 19.3.5 Status

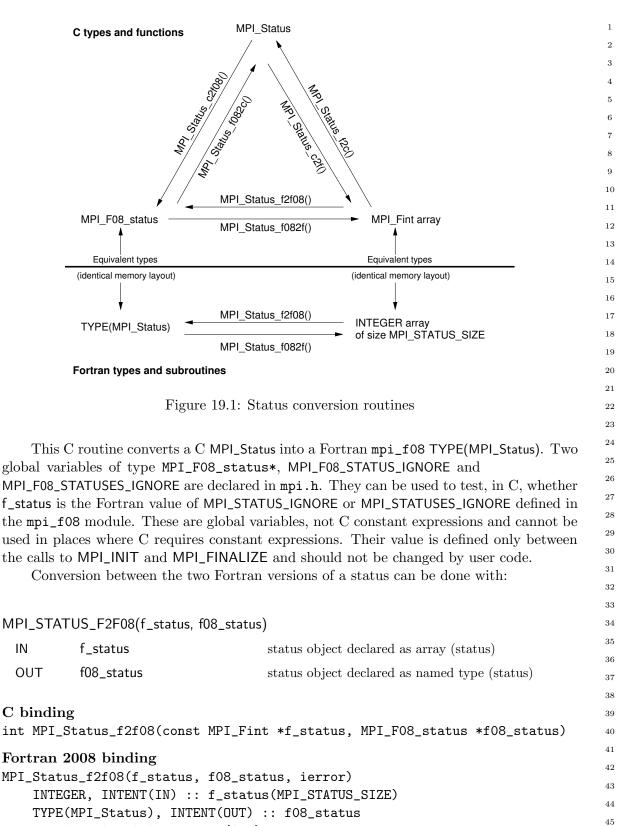
The following two procedures are provided in C to convert from a Fortran (with the mpi module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

int MPI\_Status\_f2c(const MPI\_Fint \*f\_status, MPI\_Status \*c\_status)

 $\mathbf{2}$ 

 $^{31}$ 

1 If f\_status is a valid Fortran status, but not the Fortran value of MPI\_STATUS\_IGNORE  $\mathbf{2}$ or MPI\_STATUSES\_IGNORE, then MPI\_Status\_f2c returns in c\_status a valid C status with 3 the same content. If f\_status is the Fortran value of MPI\_STATUS\_IGNORE or 4 MPI\_STATUSES\_IGNORE, or if f\_status is not a valid Fortran status, then the call is erroneous.  $\mathbf{5}$ In C, such an f\_status array can be defined with MPI\_Fint f\_status[ 6 MPI\_F\_STATUS\_SIZE]. Within this array, one can use in C the indexes MPI\_F\_SOURCE, 7MPI\_F\_TAG, and MPI\_F\_ERROR, to access the same elements as in Fortran with MPI\_SOURCE, 8 MPI\_TAG and MPI\_ERROR. The C indexes are 1 less than the corresponding indexes in 9 Fortran due to the different default array start indexes in both languages. 10 The C status has the same source, tag and error code values as the Fortran status, 11and returns the same answers when queried for count, elements, and cancellation. The 12conversion function may be called with a Fortran status argument that has an undefined 13error field, in which case the value of the error field in the C status argument is undefined. 14Two global variables of type MPI\_Fint\*, MPI\_F\_STATUS\_IGNORE and 15MPI\_F\_STATUSES\_IGNORE are declared in mpi.h. They can be used to test, in C, whether 16f\_status is the Fortran value of MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE defined in 17 the mpi module or mpif.h. These are global variables, not C constant expressions and 18 cannot be used in places where C requires constant expressions. Their value is defined only 19between the calls to MPI\_INIT and MPI\_FINALIZE and should not be changed by user code. 20To do the conversion in the other direction, we have the following: 21int MPI\_Status\_c2f(const MPI\_Status \*c\_status, MPI\_Fint \*f\_status) 2223This call converts a C status into a Fortran status, and has a behavior similar to 24MPI\_Status\_f2c. That is, the value of c\_status must not be either MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE. 2526Advice to users. There exists no separate conversion function for arrays of statuses, 27since one can simply loop through the array, converting each status with the routines 28in Figure 19.1. (End of advice to users.) 2930 *Rationale.* The handling of MPI\_STATUS\_IGNORE is required in order to layer libraries 31with only a C wrapper: if the Fortran call has passed MPI\_STATUS\_IGNORE, then the 32 C wrapper must handle this correctly. Note that this constant need not have the 33 same value in Fortran and C. If MPI\_Status\_f2c were to handle MPI\_STATUS\_IGNORE, 34 then the type of its result would have to be MPI\_Status\*\*, which was considered an 35inferior solution. (End of rationale.) 36 37 Using the mpi\_f08 Fortran module, a status is declared as TYPE(MPI\_Status). The C 38 type MPI\_F08\_status can be used to pass a Fortran TYPE(MPI\_Status) argument into a C 39 routine. Figure 19.1 illustrates all status conversion routines. Some are only available in 40 C, some in both C and the Fortran mpi and mpi\_f08 interfaces (but not in the mpif.h 41 interface). 4243 int MPI\_Status\_f082c(const MPI\_F08\_status \*f08\_status, 44MPI\_Status \*c\_status) 45This C routine converts a Fortran mpi\_f08 TYPE(MPI\_Status) into a C MPI\_Status. 4647int MPI\_Status\_c2f08(const MPI\_Status \*c\_status, 48 MPI\_F08\_status \*f08\_status)



INTEGER, OPTIONAL, INTENT(OUT) :: ierror

Fortran binding (the following procedure is not available with mpif.h) MPI\_STATUS\_F2F08(F\_STATUS, F08\_STATUS, IERROR) 46

47

1 2		ER :: F_STATUS(MPI (MPI_Status) :: F08	STATUS_SIZE), IERROR 3_STATUS
3 4 5 6		coutine converts a Fortr tran mpi_f08 TYPE(MP	ran INTEGER, DIMENSION(MPI_STATUS_SIZE) status array I_Status).
7	MPI_STAT	US_F082F(f08_status,	f_status)
8 9	IN	f08_status	status object declared as named type (status)
10 11	OUT	f_status	status object declared as array (status)
12 13 14	C binding	0	MPI_F08_status *f08_status, MPI_Fint *f_status)
15 16 17 18 19	MPI_Statu TYPE( INTEG	2008 binding ns_f082f(f08_status, (MPI_Status), INTENT GER, INTENT(OUT) :: GER, OPTIONAL, INTEN	T(IN) :: f08_status f_status(MPI_STATUS_SIZE)
20 21 22 23 24	MPI_STATU TYPE(	JS_F082F(F08_STATUS, (MPI_Status) :: F08	-
25 26		routine converts a Fort I(MPI_STATUS_SIZE) st	tran mpi_f08 TYPE(MPI_Status) into a Fortran INTEGER, tatus array.
27 28	19.3.6 M	IPI Opaque Objects	
29 30 31 32 33 34	information in the pre- language. language.	n, and have the same vious section can be u An object created in or	jects are "the same" in all languages: they carry the same meaning in both languages. The mechanism described used to pass references to MPI objects from language to ne language can be accessed, modified or freed in another
35	We ex	amine below in more o	letail issues that arise for each type of MPI object.
36 37	Datatypes		
38 39 40 41 42 43 44 45 46 47 48	MPI_TYPE defined in message set the same of formed, if language. The fu	E_GET_EXTENT will re one language is used ent will be identical to communication buffer is needed. All predefined If a datatype is commi- unction MPI_GET_ADE require that the constant	rmation in all languages. E.g., a datatype accessor like eturn the same information in all languages. If a datatype for a communication call in another language, then the the message that would be sent from the first language: s accessed, and the same representation conversion is per- d datatypes can be used in datatype constructors in any tted, it can be used for communication in any language. DRESS returns the same value in all languages. Note that nt MPI_BOTTOM have the same value in all languages (see

```
Example 19.12
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER(KIND=MPI_ADDRESS_KIND) :: AODISP(1)
! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)
/* C code */
void C_ROUTINE(MPI_Fint *ftype)
{
   int count = 5;
   int lens[2] = \{1, 1\};
   MPI_Aint displs[2];
   MPI_Datatype types[2], newtype;
   /* create an absolute datatype for buffer that consists
                                                              */
   /* of count, followed by R(5)
                                                              */
   MPI_Get_address(&count, &displs[0]);
   displs[1] = 0;
   types[0] = MPI_INT;
   types[1] = MPI_Type_f2c(*ftype);
   MPI_Type_create_struct(2, lens, displs, types, &newtype);
   MPI_Type_commit(&newtype);
   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   /* the message sent contains an int count of 5, followed
                                                              */
   /* by the 5 REAL entries of the Fortran array R.
                                                              */
}
```

Advice to implementors. The following implementation can be used: MPI addresses, 39 as returned by MPI\_GET\_ADDRESS, will have the same value in all languages. One 40 obvious choice is that MPI addresses be identical to regular addresses. The address 41 is stored in the datatype, when datatypes with absolute addresses are constructed. 42When a send or receive operation is performed, then addresses stored in a datatype 43 are interpreted as displacements that are all augmented by a base address. This base 44 address is (the address of) buf, or zero, if  $buf = MPI_BOTTOM$ . Thus, if MPI\_BOTTOM 45is zero then a send or receive call with  $buf = MPI_BOTTOM$  is implemented exactly as 46 a call with a regular buffer argument: in both cases the base address is **buf**. On the 47 other hand, if MPI\_BOTTOM is not zero, then the implementation has to be slightly 48

1

2

3

4

5

6 7

9

10

11

12

13 14

15 16

17

18

19

20

21

22 23

24

25 26

27

28

29

30

31

32 33

34

35

36

different. A test is performed to check whether  $buf = MPI_BOTTOM$ . If true, then the base address is zero, otherwise it is buf. In particular, if MPI\_BOTTOM does not have the same value in Fortran and C, then an additional test for  $buf = MPI_BOTTOM$  is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI\_BOTTOM even in C, so as to distinguish it from a NULL pointer. If MPI\_BOTTOM = c then one can still avoid the test buf = MPI\_BOTTOM, by using the displacement from MPI\_BOTTOM, i.e., the regular address - c, as the MPI address returned by MPI\_GET\_ADDRESS and stored in absolute datatypes. (*End of advice to implementors.*)

#### <sup>11</sup> 12 Callback Functions

<sup>13</sup> MPI calls may associate callback functions with MPI objects: error handlers are associated <sup>14</sup> with communicators, files, windows, and sessions; attribute copy and delete functions are <sup>15</sup> associated with attribute keys; reduce operations are associated with operation objects, etc. <sup>16</sup> In a multilanguage environment, a function passed in an MPI call in one language may be <sup>17</sup> invoked by an MPI call in another language. MPI implementations must make sure that <sup>18</sup> such invocation will use the calling convention of the language the function is bound to.

19 20

21

22

23

24 25

26

27

28

29

30

 $^{31}$ 

32 33

35

36

37

38

Advice to implementors. Callback functions need to have a language tag. This tag is set when the callback function is passed in by the library function (which is presumably different for each language and language support method), and is used to generate the right calling sequence when the callback function is invoked. (End of advice to implementors.)

Advice to users. If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI\_COMM\_NULL\_COPY\_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI\_COMM\_CREATE\_KEYVAL); see also the advice to users on page 369. (*End of advice to users.*)

<sup>34</sup> Error Handlers

Advice to implementors. Error handlers, have, in C, a variable length argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

<sup>39</sup> 40 Reduce Operations

All predefined named and unnamed datatypes as listed in Section 6.9.2 can be used in the
 listed predefined operations independent of the programming language from which the MPI
 routine is called.

Advice to users. Reduce operations receive as one of their arguments the datatype
 of the operands. Thus, one can define "polymorphic" reduce operations that work for
 C and Fortran datatypes. (End of advice to users.)

848

1

2

3

4

5

6

7

8

9

#### 19.3.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI\_TAG\_UB, MPI\_WTIME\_IS\_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI\_XXX\_CREATE\_KEYVAL call). When a communicator is duplicated, for each attribute, the corresponding copy function is called, using the right calling convention for the language of that function; and similarly, for the delete callback function.

Advice to implementors. This requires that attributes be tagged either as "C" or "Fortran" and that the language tag be checked in order to use the right calling convention for the callback function. (End of advice to implementors.)

The attribute manipulation functions described in Section 7.7 defines attributes arguments to be of type void\* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI\_ATTR\_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI\_ATTR\_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 7.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integervalued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from C, then MPI\_XXX\_get\_attr will return the address of (a pointer to) the integer-valued attribute, which is a pointer to MPI\_Aint if the attribute was stored with Fortran MPI\_XXX\_SET\_ATTR, and a pointer to int if it was stored with the deprecated Fortran MPI\_ATTR\_PUT. When an address-valued attribute is accessed from Fortran, then MPI\_XXX\_GET\_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions are used, and an integer of kind MPI\_ADDRESS\_KIND is returned. The conversion may cause truncation if deprecated attribute functions are used. In C, the deprecated routines MPI\_Attr\_put and MPI\_Attr\_get behave identical to MPI\_Comm\_set\_attr and MPI\_Comm\_get\_attr.

## Example 19.13

```
A. Setting an attribute value in C
int set_val = 3;
struct foo set_struct;
/* Set a value that is a pointer to an int */
```

1 2

3

4

5

6

 $\overline{7}$ 

8

9

10

11 12

13

14 15

16

17

18

19

20

21

22

23

 $^{24}$ 

25

26

27

28

29

30

 $^{31}$ 

32

33

34

35

36

37

38

39

40

41 42

43 44

45

 $46 \\ 47$ 

1MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval1, &set\_val); 2 /\* Set a value that is a pointer to a struct \*/ 3 MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval2, &set\_struct); 4 /\* Set an integer value \*/ 5MPI\_Comm\_set\_attr(MPI\_COMM\_WORLD, keyval3, (void \*) 17); 6 7 B. Reading the attribute value in C 8 9 int flag, \*get\_val; 10 struct foo \*get\_struct; 11 12/\* Upon successful return, get\_val == &set\_val 13(and therefore \*get\_val == 3) \*/ 14MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval1, &get\_val, &flag); 15/\* Upon successful return, get\_struct == &set\_struct \*/ 16MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval2, &get\_struct, &flag); 17/\* Upon successful return, get\_val == (void\*) 17 \*/ 18i.e., (MPI\_Aint) get\_val == 17 \*/ /\* 19MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval3, &get\_val, &flag); 2021C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 2223LOGICAL FLAG  $^{24}$ INTEGER IERR, GET\_VAL, GET\_STRUCT 2526! Upon successful return, GET\_VAL == &set\_val, possibly truncated 27CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL1, GET\_VAL, FLAG, IERR) 28! Upon successful return, GET\_STRUCT == &set\_struct, possibly truncated  $^{29}$ CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL2, GET\_STRUCT, FLAG, IERR) 30 ! Upon successful return, GET\_VAL == 17  $^{31}$ CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL3, GET\_VAL, FLAG, IERR) 32 D. Reading the attribute value with Fortran MPI-2 calls 33 34 LOGICAL FLAG 35INTEGER IERR 36 INTEGER(KIND=MPI\_ADDRESS\_KIND) GET\_VAL, GET\_STRUCT 37 38 ! Upon successful return, GET\_VAL == &set\_val 39 CALL MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, KEYVAL1, GET\_VAL, FLAG, IERR) 40! Upon successful return, GET\_STRUCT == &set\_struct 41 CALL MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, KEYVAL2, GET\_STRUCT, FLAG, IERR) 42! Upon successful return, GET\_VAL == 17 43 CALL MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, KEYVAL3, GET\_VAL, FLAG, IERR) 444546

```
Example 19.14 A. Setting an attribute value with the (deprecated) Fortran MPI-1 call
INTEGER IERR, VAL
VAL = 7
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
B. Reading the attribute value in C
int flag;
int *value;
/* Upon successful return, value points to internal MPI storage and
   *value == (int) 7 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
C. Reading the attribute value with (deprecated) Fortran MPI-1 calls
LOGICAL FLAG
INTEGER IERR, VALUE
! Upon successful return, VALUE == 7
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
D. Reading the attribute value with Fortran MPI-2 calls
LOGICAL FLAG
INTEGER IERR
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE
! Upon successful return, VALUE == 7 (sign extended)
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
Example 19.15 A. Setting an attribute value via a Fortran MPI-2 call
INTEGER IERR
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
VALUE1 = 42
VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40
CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
B. Reading the attribute value in C
int flag;
MPI_Aint *value1, *value2;
```

1

2 3

4

5

6

7 8

9

10 11

12

13

 $14 \\ 15$ 

16 17

18

19 20

21

22 23

24 25

26

27

28 29

30

31 32

33 34

35

36

37

38

39 40

41

42 43

44 45

46

1 /\* Upon successful return, value1 points to internal MPI storage and  $\mathbf{2}$ \*value1 == 42 \*/ 3 MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval1, &value1, &flag); 4 /\* Upon successful return, value2 points to internal MPI storage and 5\*value2 == 2^40 \*/ 6 MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, keyval2, &value2, &flag); 7 8 C. Reading the attribute value with (deprecated) Fortran MPI-1 calls 9 10 LOGICAL FLAG 11INTEGER IERR, VALUE1, VALUE2 1213 ! Upon successful return, VALUE1 == 42 14CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL1, VALUE1, FLAG, IERR) 15! Upon successful return, VALUE2 == 2<sup>40</sup>, or 0 if truncation 16! needed (i.e., the least significant part of the attribute word) 17CALL MPI\_ATTR\_GET(MPI\_COMM\_WORLD, KEYVAL2, VALUE2, FLAG, IERR) 18 19D. Reading the attribute value with Fortran MPI-2 calls 20LOGICAL FLAG 21INTEGER IERR 22 INTEGER(KIND=MPI\_ADDRESS\_KIND) VALUE1, VALUE2 23 $^{24}$ ! Upon successful return, VALUE1 == 42 25CALL MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, KEYVAL1, VALUE1, FLAG, IERR) 26! Upon successful return, VALUE2 == 2^40 27CALL MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, KEYVAL2, VALUE2, FLAG, IERR) 2829 The predefined MPI attributes can be integer valued or address-valued. Predefined 30 integer valued attributes, such as MPI\_TAG\_UB, behave as if they were put by a call to  $^{31}$ the deprecated Fortran routine MPI\_ATTR\_PUT, i.e., in Fortran, 32 MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, MPI\_TAG\_UB, val, flag, ierr) will return 33 in val the upper bound for tag value; in C, MPI\_Comm\_get\_attr(MPI\_COMM\_WORLD, 34MPI\_TAG\_UB, &p, &flag) will return in p a pointer to an int containing the upper bound 35 for tag value. 36 Address-valued predefined attributes, such as MPI\_WIN\_BASE behave as if they were 37 put by a C call, i.e., in Fortran, MPI\_WIN\_GET\_ATTR(win, MPI\_WIN\_BASE, val, flag, 38 ierror) will return in val the base address of the window, converted to an integer. In C, 39 MPI\_Win\_get\_attr(win, MPI\_WIN\_BASE, &p, &flag) will return in p a pointer to the window 40base, cast to (void \*). 41 42The design is consistent with the behavior specified for predefined at-Rationale. 43 tributes, and ensures that no information is lost when attributes are passed from 44language to language. Because the language interoperability for predefined attributes 45was defined based on MPI\_ATTR\_PUT, this definition is kept for compatibility reasons 46although the routine itself is now deprecated. (End of rationale.) 47 48 Advice to implementations should tag attributes either as (1) address attributes, (2) as INTEGER(KIND=MPI\_ADDRESS\_KIND) attributes or (3) as INTEGER attributes, according to whether they were set in (1) C (with MPI\_Attr\_put or MPI\_XXX\_set\_attr), (2) in Fortran with MPI\_XXX\_SET\_ATTR or (3) with the deprecated Fortran routine MPI\_ATTR\_PUT. Thus, the right choice can be made when the attribute is retrieved. (*End of advice to implementors.*)

### 19.3.8 Extra-State

Extra-state should not be modified by the copy or delete callback functions. (This is obvious from the C binding, but not obvious from the Fortran binding). However, these functions may update state that is indirectly accessed via extra-state. E.g., in C, extra-state can be a pointer to a data structure that is modified by the copy or callback functions; in Fortran, extra-state can be an index into an entry in a COMMON array that is modified by the copy or callback functions. In a multithreaded environment, users should be aware that distinct threads may invoke the same callback function concurrently: if this function modifies state associated with extra-state, then mutual exclusion code must be used to protect updates and accesses to the shared state.

#### 19.3.9 Constants

MPI constants have the same value in all languages, unless specified otherwise. This does not apply to constant handles (MPI\_INT, MPI\_COMM\_WORLD, MPI\_ERRORS\_RETURN, MPI\_SUM, etc.) These handles need to be converted, as explained in Section 19.3.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

Advice to users. This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

char name [MPI\_MAX\_OBJECT\_NAME];

(End of advice to users.)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI\_BOTTOM or MPI\_STATUS\_IGNORE may have different values in different languages.

Rationale. The current MPI standard specifies that MPI\_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI\_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take  $MPI_BOTTOM = 0$  (Caveat: Defining  $MPI_BOTTOM =$ 0 implies that NULL pointer cannot be distinguished from MPI\_BOTTOM; it may be that MPI\_BOTTOM = 1 is better. See the advice to implementors in the *Datatypes* subsection in Section 19.3.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (End of rationale.)

 $^{24}$ 

## 19.3.10 Interlanguage Communication The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI\_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI\_BYTE or MPI\_PACKED. Interlanguage communication is allowed if it complies with these rules. **Example 19.16** In the example below, a Fortran array is sent from Fortran and received in C. ! FORTRAN CODE SUBROUTINE MYEXAMPLE() USE mpi\_f08 REAL :: R(5)INTEGER :: IERR, MYRANK, AOBLEN(1) TYPE(MPI\_Datatype) :: TYPE, AOTYPE(1) INTEGER(KIND=MPI\_ADDRESS\_KIND) :: AODISP(1) ! create an absolute datatype for array R AOBLEN(1) = 5CALL MPI\_GET\_ADDRESS(R, AODISP(1), IERR) AOTYPE(1) = MPI\_REAL CALL MPI\_TYPE\_CREATE\_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR) CALL MPI\_TYPE\_COMMIT(TYPE, IERR) CALL MPI\_COMM\_RANK(MPI\_COMM\_WORLD, MYRANK, IERR) IF (MYRANK.EQ.O) THEN CALL MPI\_SEND(MPI\_BOTTOM, 1, TYPE, 1, 0, MPI\_COMM\_WORLD, IERR) ELSE CALL C\_ROUTINE(TYPE%MPI\_VAL) END IF END SUBROUTINE /\* C code \*/ void C\_ROUTINE(MPI\_Fint \*fhandle) { MPI\_Datatype type; MPI\_Status status; type = MPI\_Type\_f2c(\*fhandle); MPI\_Recv(MPI\_BOTTOM, 1, type, 0, 0, MPI\_COMM\_WORLD, &status);

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e.,

1 2 3

4

5

6

7

8 9

10

11 12

13

14

15

16

17

18

19 20

21

22

23

 $^{24}$ 

25

26 27

28

29

30

 $^{31}$ 

32

33

34

35 36 37

38

39

40

41 42

43

44

 $45 \\ 46$ 

47

48

}

if the Fortran type INTEGER is identical to the C type int, then an MPI implementation may allow data to be sent with datatype MPI\_INTEGER and be received with datatype MPI\_INT. However, such code is not portable.

 $\mathbf{2}$ 

## Annex A

# Language Bindings Summary

In this section we summarize the specific bindings for C and Fortran. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

## A.1 Defined Values and Handles

### A.1.1 Defined Constants

The C and Fortran names are listed below. Constants with the type const int may also be implemented as literal integer constants substituted by the preprocessor.

Error classes	24 25
C type: const int (or unnamed enum)	26
Fortran type: INTEGER	27
MPI_SUCCESS	28
MPI_ERR_BUFFER	29
MPI_ERR_COUNT	30
MPI_ERR_TYPE	31
MPI_ERR_TAG	32
MPI_ERR_COMM	33
MPI_ERR_RANK	34
MPI_ERR_REQUEST	35
MPI_ERR_ROOT	36
MPI_ERR_GROUP	37
MPI_ERR_OP	38
MPI_ERR_TOPOLOGY	39
MPI_ERR_DIMS	40
MPI_ERR_ARG	41
MPI_ERR_UNKNOWN	42
MPI_ERR_TRUNCATE	43
MPI_ERR_OTHER	44
MPI_ERR_INTERN	45
MPI_ERR_PENDING	46
(Continued on next page)	47
	48

Error classes (continued)
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_ERR_IN_STATUS
MPI_ERR_ACCESS
MPI_ERR_AMODE
MPI_ERR_ASSERT
MPI_ERR_BAD_FILE
MPI_ERR_BASE
MPI_ERR_CONVERSION
MPI_ERR_DISP
MPI_ERR_DUP_DATAREP
MPI_ERR_FILE_EXISTS
MPI_ERR_FILE_IN_USE
MPI_ERR_FILE
MPI_ERR_INFO_KEY
MPI_ERR_INFO_NOKEY
MPI_ERR_INFO_VALUE
MPI_ERR_INFO
MPI_ERR_IO
MPI_ERR_KEYVAL
MPI_ERR_LOCKTYPE
MPI_ERR_NAME
MPI_ERR_NO_MEM
MPI_ERR_NOT_SAME
MPI_ERR_NO_SPACE
MPI_ERR_NO_SUCH_FILE
MPI_ERR_PORT
MPI_ERR_PROC_ABORTED
MPI_ERR_QUOTA
MPI_ERR_READ_ONLY
MPI_ERR_RMA_ATTACH
MPI_ERR_RMA_CONFLICT
MPI_ERR_RMA_RANGE
MPI_ERR_RMA_SHARED
MPI_ERR_RMA_SYNC
MPI_ERR_RMA_FLAVOR
MPI_ERR_SERVICE
MPI_ERR_SESSION
MPI_ERR_SIZE
MPI_ERR_SPAWN
MPI_ERR_UNSUPPORTED_DATAREP
MPI_ERR_UNSUPPORTED_OPERATION
MPI_ERR_VALUE_TOO_LARGE
MPI_ERR_WIN
(Continued on next page)

 $^{48}$ 

	Error classes (continued)	1
	C type: const int (or unnamed enum)	2
	Fortran type: INTEGER	3
	MPI_T_ERR_CANNOT_INIT	4
	MPI_T_ERR_NOT_ACCESSIBLE	5
	MPI_T_ERR_NOT_INITIALIZED	6
	MPI_T_ERR_NOT_SUPPORTED	7
	MPI_T_ERR_MEMORY	8
	MPI_T_ERR_INVALID	9
	MPI_T_ERR_INVALID_INDEX	10
	MPI_T_ERR_INVALID_ITEM	11
	MPI_T_ERR_INVALID_SESSION	12
	MPI_T_ERR_INVALID_HANDLE	13
	MPI_T_ERR_INVALID_NAME	14
	MPI_T_ERR_OUT_OF_HANDLES	15
	MPI_T_ERR_OUT_OF_SESSIONS	16
	MPI_T_ERR_CVAR_SET_NOT_NOW	17
	MPI_T_ERR_CVAR_SET_NEVER	18
	MPI_T_ERR_PVAR_NO_WRITE	19
	MPI_T_ERR_PVAR_NO_STARTSTOP	20
	MPI_T_ERR_PVAR_NO_ATOMIC	21
	MPI_ERR_LASTCODE	22
	Buffer Address Constants	23 24
C type: void * $\alpha$		25
	edefined memory location) <sup>1</sup>	26
MPI_BOTTOM		27
MPI_IN_PLACE		28
	Fortran these constants are not usable for initialization	29
expressions or	assignment. See Section 2.5.4.	30
	Assorted Constants	31
	C type: const int (or unnamed enum)	32 33
	Fortran type: INTEGER	
	MPI_PROC_NULL	34 35
	MPI_ANY_SOURCE	35
	MPI_ANY_TAG	36 37
	MPI_UNDEFINED	37
	MPI_BSEND_OVERHEAD	38
	MPI_KEYVAL_INVALID	40
	MPI_LOCK_EXCLUSIVE	40
	MPI_LOCK_SHARED	41 42
	MPI_ROOT	42
		43 44
	No Process Message Handle	44 45
	type: MPI_Message	46
	rtran type: INTEGER or TYPE(MPI_Message)	47
	PI_MESSAGE_NO_PROC	48

	Fortran Support Method Specific Constants
	Fortran type: LOGICAL
	MPI_SUBARRAYS_SUPPORTED (Fortran only)
	MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)
$\mathbf{S}$	Status array size and reserved index values (Fortran only)
Fe	ortran type: INTEGER
Μ	IPI_STATUS_SIZE
Μ	IPI_SOURCE
	- IPI_TAG
	_ IPI_ERROR
Fo	ortran status array size and reserved index values (C only)
C t	type: int
	PI_F_STATUS_SIZE
MF	PI_F_SOURCE
MF	PI_F_TAG
MF	PI_F_ERROR
	Variable Address Size (Fortran only)
	Fortran type: INTEGER
	MPI_ADDRESS_KIND
	MPI_COUNT_KIND
	MPI_INTEGER_KIND
	MPI_OFFSET_KIND
	Error-handling specifiers
	C type: MPI_Errhandler
	Fortran type: INTEGER or TYPE(MPI_Errhandler)
	MPI_ERRORS_ARE_FATAL
	MPI_ERRORS_ABORT
	MPI_ERRORS_RETURN
	Maximum Sizes for Strings
	Maximum Sizes for Strings
	Maximum Sizes for Strings         C type: const int (or unnamed enum)
	Maximum Sizes for Strings C type: const int (or unnamed enum) Fortran type: INTEGER
	Maximum Sizes for Strings         C type: const int (or unnamed enum)         Fortran type: INTEGER         MPI_MAX_DATAREP_STRING
	Maximum Sizes for StringsC type: const int (or unnamed enum)Fortran type: INTEGERMPI_MAX_DATAREP_STRINGMPI_MAX_ERROR_STRING
	Maximum Sizes for StringsC type: const int (or unnamed enum)Fortran type: INTEGERMPI_MAX_DATAREP_STRINGMPI_MAX_ERROR_STRINGMPI_MAX_INFO_KEY
	Maximum Sizes for StringsC type: const int (or unnamed enum)Fortran type: INTEGERMPI_MAX_DATAREP_STRINGMPI_MAX_ERROR_STRINGMPI_MAX_INFO_KEYMPI_MAX_INFO_VAL
	Maximum Sizes for StringsC type: const int (or unnamed enum)Fortran type: INTEGERMPI_MAX_DATAREP_STRINGMPI_MAX_ERROR_STRINGMPI_MAX_INFO_KEYMPI_MAX_LINFO_VALMPI_MAX_LIBRARY_VERSION_STRING
	Maximum Sizes for Strings C type: const int (or unnamed enum) Fortran type: INTEGER MPI_MAX_DATAREP_STRING MPI_MAX_ERROR_STRING MPI_MAX_INFO_KEY MPI_MAX_INFO_VAL MPI_MAX_LIBRARY_VERSION_STRING MPI_MAX_OBJECT_NAME
	Maximum Sizes for StringsC type: const int (or unnamed enum)Fortran type: INTEGERMPI_MAX_DATAREP_STRINGMPI_MAX_ERROR_STRINGMPI_MAX_INFO_KEYMPI_MAX_INFO_VALMPI_MAX_LIBRARY_VERSION_STRINGMPI_MAX_OBJECT_NAMEMPI_MAX_PORT_NAME
	Maximum Sizes for StringsC type: const int (or unnamed enum)Fortran type: INTEGERMPI_MAX_DATAREP_STRINGMPI_MAX_ERROR_STRINGMPI_MAX_INFO_KEYMPI_MAX_INFO_VALMPI_MAX_LIBRARY_VERSION_STRINGMPI_MAX_OBJECT_NAMEMPI_MAX_PORT_NAMEMPI_MAX_PROCESSOR_NAME
	Maximum Sizes for StringsC type: const int (or unnamed enum)Fortran type: INTEGERMPI_MAX_DATAREP_STRINGMPI_MAX_ERROR_STRINGMPI_MAX_INFO_KEYMPI_MAX_INFO_VALMPI_MAX_LIBRARY_VERSION_STRINGMPI_MAX_OBJECT_NAMEMPI_MAX_PORT_NAME

Named Predefined Datatypes	C types	
C type: MPI_Datatype		
Fortran type: INTEGER		
or TYPE(MPI_Datatype)		
MPI_CHAR	char	
	(treated as printable character)	
MPI_SHORT	signed short int	
MPI_INT	signed int	
MPI_LONG	signed long	
MPI_LONG_LONG_INT	signed long long	
MPI_LONG_LONG (as a synonym)	signed long long	
MPI_SIGNED_CHAR	signed char	
	(treated as integral value)	
MPI_UNSIGNED_CHAR	unsigned char	
	(treated as integral value)	
MPI_UNSIGNED_SHORT	unsigned short	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long	
MPI_UNSIGNED_LONG_LONG	unsigned long long	
MPI_FLOAT	float	
MPI_DOUBLE	double	
MPI_LONG_DOUBLE	long double	
MPI_WCHAR	wchar_t	
	(defined in <stddef.h>)</stddef.h>	
	(treated as printable character)	
MPI_C_BOOL	_Bool	
MPI_INT8_T	int8_t	
MPI_INT16_T	int16_t	
MPI_INT32_T	int32_t	
MPI_INT64_T	int64_t	
MPI_UINT8_T	uint8_t	
MPI_UINT16_T	uint16_t	
MPI_UINT32_T	uint32_t	
MPI_UINT64_T	uint64_t	
MPI_AINT	MPI_Aint	
MPI_COUNT	MPI_Count	
MPI_OFFSET	MPI_Offset	
MPI_C_COMPLEX	float _Complex	
MPI_C_FLOAT_COMPLEX	float _Complex	
MPI_C_DOUBLE_COMPLEX	double _Complex	
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex	
MPI_BYTE	(any C type)	
MPI_PACKED	(any C type)	

l Predefined Datatypes | C types

	Named Predefined Datatypes	Fortra	n types
-	C type: MPI_Datatype		
	Fortran type: INTEGER		
	or TYPE(MPI_Datatype)		
-	MPI_INTEGER	INTEGE	2
	MPI_REAL	REAL	
	MPI_DOUBLE_PRECISION	DOUBLE	PRECISION
	MPI_COMPLEX	COMPLEX	ζ
	MPI_LOGICAL	LOGICA	
	MPI_CHARACTER	CHARAC	TER(1)
	MPI_AINT	INTEGE	R(KIND=MPI_ADDRESS_KIND)
	MPI_COUNT		R(KIND=MPI_COUNT_KIND)
	MPI_OFFSET		R(KIND=MPI_OFFSET_KIND)
	MPI_BYTE	(any Fortran type)	
	MPI_PACKED		rtran type)
-		(any ro	itian type)
	Named Predefined Datatypes	$\mathbf{s}^1 \mid \mathbf{C} +$	+ types
	C type: MPI_Datatype		
	Fortran type: INTEGER		
	or TYPE(MPI_Datatype)		
	MPI_CXX_BOOL	boo	1
	MPI_CXX_FLOAT_COMPLEX		::complex <float></float>
	MPI_CXX_DOUBLE_COMPLEX		::complex <double></double>
	MPI_CXX_LONG_DOUBLE_COMPLE		::complex <long double=""></long>
	$\frac{1}{1}$ If an accompanying C++ compi		
	MPI datatypes in this table are n		0,
		not donn	
	Optional datatypes (Fo	ortran)	Fortran types
	C type: MPI_Datatype	,	
	Fortran type: INTEGER		
	Fortran type: INTEGER or TYPE(MPI_Datatype)		
	01		DOUBLE COMPLEX
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX		DOUBLE COMPLEX INTEGER*1
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1		INTEGER*1
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2		INTEGER*1 INTEGER*2
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4		INTEGER*1 INTEGER*2 INTEGER*4
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL8		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL8 MPI_REAL16		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4 MPI_COMPLEX8		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4 COMPLEX*8
	or TYPE(MPI_Datatype) MPI_DOUBLE_COMPLEX MPI_INTEGER1 MPI_INTEGER2 MPI_INTEGER4 MPI_INTEGER8 MPI_INTEGER16 MPI_REAL2 MPI_REAL4 MPI_REAL4 MPI_REAL16 MPI_COMPLEX4		INTEGER*1 INTEGER*2 INTEGER*4 INTEGER*8 INTEGER*16 REAL*2 REAL*4 REAL*8 REAL*16 COMPLEX*4

Datatypes for reduction functions (C)	1
C type: MPI_Datatype	2
Fortran type: INTEGER or TYPE(MPI_Datatype)	3
MPI_FLOAT_INT	4
MPI_DOUBLE_INT	5
MPI_LONG_INT	6
MPI_2INT	7
	8
MPI_LONG_DOUBLE_INT	9
Datatypes for reduction functions (Fortran)	10
C type: MPI_Datatype	11
Fortran type: INTEGER or TYPE(MPI_Datatype)	12
MPI_2REAL	13
MPI_2DOUBLE_PRECISION	14
MPI_2INTEGER	15
	16
Reserved communicators	17
C type: MPI_Comm	18
Fortran type: INTEGER or TYPE(MPI_Comm)	19
MPI_COMM_WORLD	20
MPI_COMM_SELF	21
	22
Communicator split type constants	23
C type: const int (or unnamed enum)	24
Fortran type: INTEGER	25
MPI_COMM_TYPE_SHARED	26
MPI_COMM_TYPE_HW_UNGUIDED	27
MPI_COMM_TYPE_HW_GUIDED	28
Results of communicator and group compariso	ns 30
C type: const int (or unnamed enum)	31
Fortran type: INTEGER	32
MPI_IDENT	33
MPI_CONGRUENT	34
MPI_SIMILAR	35
MPI_UNEQUAL	36
Environmental inquiry info key	37
C type: MPI_Info	38
Fortran type: INTEGER or TYPE(MPI_Info)	39
MPI_INFO_ENV	40
Environmental inquiry keys	41
C type: const int (or unnamed enum)	43
Fortran type: INTEGER	44
MPI_TAG_UB	45
MPI_TAG_OD MPI_IO	46
	47
	48
MPI_WTIME_IS_GLOBAL	10

L	Collective Operations
2	C type: MPI_Op
	Fortran type: INTEGER or TYPE(MPI_Op)
	MPI_MAX
	MPI_MIN
	MPI_SUM
	MPI_MAXLOC
	MPI_MINLOC
	MPI_BAND
	MPI_BOR
	MPI_BXOR
	MPI_LAND
	MPI_LOR
	MPI_LXOR
	MPI_REPLACE
	MPI_NO_OP
	Null Handles
	C/Fortran name
	C type / Fortran type
	MPI_GROUP_NULL
	MPI_Group / INTEGER or TYPE(MPI_Group)
	MPI_COMM_NULL
	MPI_Comm / INTEGER or TYPE(MPI_Comm)
	MPI_DATATYPE_NULL
	MPI_Datatype / INTEGER or TYPE(MPI_Datatype)
	MPI_REQUEST_NULL
	$MPI\_Request \ / \ \mathtt{INTEGER} \ \mathrm{or} \ TYPE(MPI\_Request)$
	MPI_OP_NULL
	MPI_Op / INTEGER or TYPE(MPI_Op)
	MPI_ERRHANDLER_NULL
	MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler)
	MPI_FILE_NULL
	MPI_File / INTEGER or TYPE(MPI_File)
	MPI_INFO_NULL
	MPI_Info / INTEGER or TYPE(MPI_Info)
	MPI_SESSION_NULL
	MPI_Session / INTEGER or TYPE(MPI_Session)
	MPI_Win / INTEGER or TYPE(MPI_Win)
	MPI_MESSAGE_NULL
	MPI_Message / INTEGER or TYPE(MPI_Message)
	Empty group
	C tarmer MDL Current
	C type: MPI_Group
	C type: MPI_Group Fortran type: INTEGER or TYPE(MPI_Group) MPI_GROUP_EMPTY

	Topologies
	C type: const int (or unnamed enum)
	Fortran type: INTEGER
	MPI_GRAPH
	MPI_CART
	MPI_DIST_GRAPH
	Predefined functions
C/Fortran name	1 redefined functions
C type	
/ Fortran type with mp	i module / Fortran type with mpi_f08 module
MPI_COMM_NULL_CO	, ,
MPI_Comm_copy_attr_:	
/ COMM_COPY_ATTR_FUN	_
MPI_COMM_DUP_FN	/
MPI_Comm_copy_attr_:	function
/ COMM_COPY_ATTR_FUN	$  $ CTION / PROCEDURE(MPI_Comm_copy_attr_function) $ $
MPI_COMM_NULL_DEI	_ETE_FN
MPI_Comm_delete_att:	
/ COMM_DELETE_ATTR_F	, , , , , , , , , , , , , , , , , , , ,
MPI_WIN_NULL_COPY	
MPI_Win_copy_attr_f	
/ WIN_COPY_ATTR_FUNC	$TION / PROCEDURE(MPI_Win_copy_attr_function)$
MPI_WIN_DUP_FN	
MPI_Win_copy_attr_f	
/ WIN_COPY_ATTR_FUNC	, ,
MPI_WIN_NULL_DELET	
MPI_Win_delete_attr	
/ WIN_DELETE_ATTR_FU	· · · · · · · · · · · · · · · · · · ·
MPI_TYPE_NULL_COP	—
MPI_Type_copy_attr_:	_
/ TYPE_COPY_ATTR_FUN	<pre>ICTION / PROCEDURE(MPI_Type_copy_attr_function) 1)</pre>
MPI_TYPE_DUP_FN	
MPI_Type_copy_attr_: / TYPE_COPY_ATTR_FUN	_
MPI_TYPE_NULL_DEL	, , , , , , , , , , , , , , , , , , , ,
MPI_Type_delete_att:	
/ TYPE_DELETE_ATTR_F	
MPI_CONVERSION_FN	
MPI_Datarep_convers	
/ DATAREP_CONVERSION	
MPI_CONVERSION_FN	, ,
MPI_Datarep_convers	
-	RE(MPI_Datarep_conversion_function_c)
/ (n/a) / PROCEDUF	
	ementors (on page 369) and advice to users (on page 369)
See the advice to impl	ementors (on page 369) and advice to users (on page 369) ctran functions MPI_COMM_NULL_COPY_FN, in

Deprecated predefined functions
C/Fortran name
C type / Fortran type with mpi module
MPI_NULL_COPY_FN
MPI_Copy_function / COPY_FUNCTION
MPI_DUP_FN
MPI_Copy_function / COPY_FUNCTION
MPI_NULL_DELETE_FN
MPI_Delete_function / DELETE_FUNCTION
<u>.</u>
Predefined Attribute Keys
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_APPNUM
MPI_LASTUSEDCODE
MPI_UNIVERSE_SIZE
MPI_WIN_BASE
MPI_WIN_DISP_UNIT
MPI_WIN_SIZE
MPI_WIN_CREATE_FLAVOR
MPI_WIN_MODEL
MPI Window Create Flavors
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_WIN_FLAVOR_CREATE
MPI_WIN_FLAVOR_ALLOCATE
MPI_WIN_FLAVOR_DYNAMIC
MPI_WIN_FLAVOR_SHARED
MPI Window Models
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_WIN_SEPARATE
MPI_WIN_UNIFIED

Mode Constants
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_MODE_APPEND
MPI_MODE_CREATE
MPI_MODE_DELETE_ON_CLOSE
MPI_MODE_EXCL
MPI_MODE_NOCHECK
MPI_MODE_NOPRECEDE
MPI_MODE_NOPUT
MPI_MODE_NOSTORE
MPI_MODE_NOSUCCEED
MPI_MODE_RDONLY
MPI_MODE_RDWR
MPI_MODE_SEQUENTIAL
MPI_MODE_UNIQUE_OPEN
MPI_MODE_WRONLY
Datatype Decoding Constants
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_COMBINER_CONTIGUOUS
MPI_COMBINER_DARRAY
MPI_COMBINER_DUP
MPI_COMBINER_F90_COMPLEX
MPI_COMBINER_F90_INTEGER
MPI_COMBINER_F90_REAL
MPI_COMBINER_HINDEXED
MPI_COMBINER_HVECTOR
MPI_COMBINER_INDEXED_BLOCK
MPI_COMBINER_HINDEXED_BLOCK
MPI_COMBINER_INDEXED
MPI_COMBINER_NAMED
MPI_COMBINER_RESIZED
MPI_COMBINER_STRUCT
MPI_COMBINER_SUBARRAY
MPI_COMBINER_VECTOR
Threads Constants
C type: const int (or unnamed enum)
Fortran type: INTEGER
MPI_THREAD_FUNNELED
MPI_THREAD_MULTIPLE
MPI_THREAD_SERIALIZED
MPI_THREAD_SINGLE

1	File Operation Constants, Part 1
2	C type: const MPI_Offset (or unnamed enum)
3	Fortran type: INTEGER(KIND=MPI_OFFSET_KIND)
4	MPI_DISPLACEMENT_CURRENT
5	
6	File Operation Constants, Part 2
7	C type: const int (or unnamed enum)
8	Fortran type: INTEGER
9	MPI_DISTRIBUTE_BLOCK
10	MPI_DISTRIBUTE_CYCLIC
11	MPI_DISTRIBUTE_DFLT_DARG
12	MPI_DISTRIBUTE_NONE
13	MPI_ORDER_C
14	MPI_ORDER_FORTRAN
15	MPI_SEEK_CUR
16	MPI_SEEK_END
17	MPI_SEEK_SET
18	
19	F90 Datatype Matching Constants
20	C type: const int (or unnamed enum)
21	Fortran type: INTEGER
22	MPI_TYPECLASS_COMPLEX
23	MPI_TYPECLASS_INTEGER
24	MPI_TYPECLASS_REAL
25	
26	Constants Specifying Empty or Ignored Input
27	C/Fortran name
28	C type / Fortran type <sup>1</sup>
29	MPI_ARGVS_NULL
30	char*** / 2-dim. array of CHARACTER*(*)
31	MPI_ARGV_NULL
32 33	char** / array of CHARACTER*(*)
34	MPI_ERRCODES_IGNORE
35	int* / INTEGER array
36	MPI_STATUSES_IGNORE
37	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)
38	or TYPE(MPI_Status), DIMENSION(*)
39	MPI_STATUS_IGNORE
40	MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)
41	or TYPE(MPI_Status)
42	MPI_UNWEIGHTED int* / INTEGER array
43	MPI_WEIGHTS_EMPTY
44	
45	$\frac{\text{int}* / \text{INTEGER array}}{1}$ Note that in Fortran these constants are not usable for initialization
46	expressions or assignment. See Section 2.5.4.
47	CAPTERSTOTE OF ARRESTOTETTE OF A CONTRACT $4.0.4$
48	

C Constants specify	ing Ignored Input (no Fortran)	1
C type: MPI_Fint*	equivalent to Fortran	2
MPI_F_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi / mpif.h	3
MPI_F_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi / mpif.h	4
C type: MPI_F08_status*	equivalent to Fortran	5
MPI_F08_STATUSES_IGNORE	MPI_STATUSES_IGNORE in mpi_f08	6
MPI_F08_STATUS_IGNORE	MPI_STATUS_IGNORE in mpi_f08	7
		8
C preprocessor Con	stants and Fortran Parameters	9
C type: C-preprocessor n	nacro that expands to an int value	10
Fortran type: INTEGER		11
MPI_SUBVERSION		12
MPI_VERSION		13
		14
	he MPI tool information interface	15
MPI_T_ENUM_NULL		16
MPI_T_enum		17
MPI_T_CVAR_HANDLE_NU	ILL	18
$MPI_T_cvar_handle$		19
MPI_T_PVAR_HANDLE_NU	ILL	20
$MPI_T_pvar_handle$		21
MPI_T_PVAR_SESSION_NU	ILL	22
MPI_T_pvar_session		23
<b>X71</b>		24 25
	ne MPI tool information interface	26
C type: const int (or unn	,	27
MPI_T_VERBOSITY_USER		
		28
	2_DETAIL	28 29
MPI_T_VERBOSITY_USER	2_DETAIL 2_ALL	
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE	2_DETAIL 2_ALL ER_BASIC	29
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_TUNI	2_DETAIL 2_ALL ER_BASIC ER_DETAIL	29 30
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_TUNI	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL	29 30 31
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC	29 30 31 32
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_TUNI MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38 39
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38 39 40
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38 39 40 41
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38 39 40 41
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
MPI_T_VERBOSITY_USER MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_TUNE MPI_T_VERBOSITY_MPIC MPI_T_VERBOSITY_MPIC	2_DETAIL 2_ALL ER_BASIC ER_DETAIL ER_ALL PEV_BASIC PEV_DETAIL	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

870	ANNEX A. LANGUAGE BINDINGS SUMM.	ARY
1	Constants to identify associations of variables	
2	in the MPI tool information interface	
3	C type: const int (or unnamed enum)	
4	MPI_T_BIND_NO_OBJECT	
5	MPI_T_BIND_MPI_COMM	
6	MPI_T_BIND_MPI_DATATYPE	
7	MPI_T_BIND_MPI_ERRHANDLER	
8	MPI_T_BIND_MPI_FILE	
9	MPI_T_BIND_MPI_GROUP	
10	MPI_T_BIND_MPI_OP	
11	MPI_T_BIND_MPI_REQUEST	
12		
13	MPI_T_BIND_MPI_WIN	
14	MPI_T_BIND_MPI_MESSAGE	
15	MPI_T_BIND_MPI_INFO	
16	MPI_T_BIND_MPI_SESSION	
17	Constants describing the same of a control variable	
18	Constants describing the scope of a control variable in the MPI tool information interface	
19		
	C type: const int (or unnamed enum)	
20	MPI_T_SCOPE_CONSTANT	
21	MPI_T_SCOPE_READONLY	
22	MPI_T_SCOPE_LOCAL	
23	MPI_T_SCOPE_GROUP	
24	MPI_T_SCOPE_GROUP_EQ	
25	MPI_T_SCOPE_ALL	
26	MPI_T_SCOPE_ALL_EQ	
27		
28	Additional constants used	
29 30	by the MPI tool information interface	
31	C type: MPI_T_pvar_handle	
32	MPI_T_PVAR_ALL_HANDLES	
33		
34	Performance variables classes used by the	
35	MPI tool information interface	
36	C type: const int (or unnamed enum)	
37	MPI_T_PVAR_CLASS_STATE	
38	MPI_T_PVAR_CLASS_LEVEL	
39	MPI_T_PVAR_CLASS_SIZE	
40	MPI_T_PVAR_CLASS_PERCENTAGE	
	MPI_T_PVAR_CLASS_HIGHWATERMARK	
41	MPI_T_PVAR_CLASS_LOWWATERMARK	
42	MPI_T_PVAR_CLASS_COUNTER	
43	MPI_T_PVAR_CLASS_AGGREGATE	
44	MPI_T_PVAR_CLASS_TIMER	
45 46	MPI_T_PVAR_CLASS_GENERIC	
40		

	Source event ordering guarantees in the	1
	MPI tool information interface	2 3
	C type: MPI_T_source_order	4
	MPI_T_SOURCE_ORDERED	5
	MPI_T_SOURCE_UNORDERED	6
	Callback safety requirement levels used in the MPI tool information interface	7
	C type: MPI_T_cb_safety	- 9
	MPI_T_CB_REQUIRE_NONE	- 10
	MPI_T_CB_REQUIRE_MONE MPI_T_CB_REQUIRE_MPI_RESTRICTED	11
	MPI_T_CB_REQUIRE_THREAD_SAFE	12
	MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE	13
		- 14
A.1.2 Types		15
The fellowing an	a defined C time definitions included in the flarmin	16
The following are	e defined C type definitions included in the file mpi.h.	17
/* C opaque ty	rpes */	18
MPI_Aint		19
MPI_Count		20
MPI_Fint		21
MPI_Offset		22
MPI_Status		23
MPI_F08_status		24
		25
/* C handles t	o assorted structures */	26
MPI_Comm		27
MPI_Datatype		28
MPI_Errhandler		29
MPI_File		30
MPI_Group		31
MPI_Info		32
MPI_Message		33
MPI_Op		34
MPI_Request		35 36
MPI_Session		30
MPI_Win		38
/* Types for t	the MPI_T interface */	39
MPI_T_enum		40
MPI_T_cvar_handl	le	41
MPI_T_pvar_hand	le	42
MPI_T_pvar_session	on	43
MPI_T_event_insta	ance	44
MPI_T_event_regis	stration	45
MPI_T_source_ord	ler	46
MPI_T_cb_safety		47
		48

12 3 The following are defined Fortran type definitions included in the mpi\_f08 and mpi 4 modules. 5! Fortran opaque types in the mpi\_f08 and mpi modules 6 TYPE(MPI\_Status) 7 8 ! Fortran handles in the mpi\_f08 and mpi modules 9 TYPE(MPI\_Comm) 10 TYPE(MPI\_Datatype) 11 TYPE(MPI\_Errhandler) 12TYPE(MPI\_File) 13 TYPE(MPI\_Group) 14 TYPE(MPI\_Info) 15TYPE(MPI\_Message) 16TYPE(MPI\_Op) 17TYPE(MPI\_Request) 18 TYPE(MPI\_Session) 19 TYPE(MPI\_Win) 2021A.1.3 Prototype Definitions 22 23C Bindings 24The following are defined C typedefs for user-defined functions, also included in the file 25mpi.h. 2627/\* prototypes for user-defined functions \*/ 28typedef void MPI\_User\_function(void \*invec, void \*inoutvec, int \*len, 29MPI\_Datatype \*datatype); 30  $^{31}$ typedef void MPI\_User\_function\_c(void \*invec, void \*inoutvec, 32 MPI\_Count \*len, MPI\_Datatype \*datatype); 33 34 typedef int MPI\_Comm\_copy\_attr\_function(MPI\_Comm oldcomm, int comm\_keyval, void \*extra\_state, void \*attribute\_val\_in, 35 void \*attribute\_val\_out, int \*flag); 36 37 typedef int MPI\_Comm\_delete\_attr\_function(MPI\_Comm comm, int comm\_keyval, 38 void \*attribute\_val, void \*extra\_state); 3940typedef int MPI\_Win\_copy\_attr\_function(MPI\_Win oldwin, int win\_keyval, 41 void \*extra\_state, void \*attribute\_val\_in, 42void \*attribute\_val\_out, int \*flag); 43 typedef int MPI\_Win\_delete\_attr\_function(MPI\_Win win, int win\_keyval, 44 void \*attribute\_val, void \*extra\_state); 4546typedef int MPI\_Type\_copy\_attr\_function(MPI\_Datatype oldtype, 47int type\_keyval, void \*extra\_state, void \*attribute\_val\_in, 48 void \*attribute\_val\_out, int \*flag);

typedef	<pre>int MPI_Type_delete_attr_function(MPI_Datatype datatype,</pre>	1 2
typedef	<pre>void MPI_Comm_errhandler_function(MPI_Comm *comm, int *error_code, );</pre>	3 4 5
typedef	<pre>void MPI_Win_errhandler_function(MPI_Win *win, int *error_code, );</pre>	6 7 8
typedef	<pre>void MPI_File_errhandler_function(MPI_File *file, int *error_code, );</pre>	9 10
typedef	<pre>void MPI_Session_errhandler_function(MPI_Session *session,</pre>	11 12
typedef	<pre>int MPI_Grequest_query_function(void *extra_state,</pre>	13 14 15
typedef	<pre>int MPI_Grequest_free_function(void *extra_state);</pre>	16
typedef	<pre>int MPI_Grequest_cancel_function(void *extra_state, int complete);</pre>	17 18
typedef	<pre>int MPI_Datarep_extent_function(MPI_Datatype datatype,</pre>	19 20 21
typedef	<pre>int MPI_Datarep_conversion_function(void *userbuf, MPI_Datatype datatype, int count, void *filebuf, MPI_Offset position, void *extra_state);</pre>	22 23 24
typedef	<pre>int MPI_Datarep_conversion_function_c(void *userbuf, MPI_Datatype datatype, MPI_Count count, void *filebuf, MPI_Offset position, void *extra_state);</pre>	25 26 27 28
typedef	<pre>void MPI_T_event_cb_function(MPI_T_event_instance event_instance, MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, void *user_data);</pre>	29 30 31
typedef	<pre>void MPI_T_event_free_cb_function(     MPI_T_event_registration event_registration,     MPI_T_cb_safety cb_safety, void *user_data);</pre>	32 33 34 35
typedef	<pre>void MPI_T_event_dropped_cb_function(MPI_Count count,</pre>	36 37 38 39
Fortran 2	008 Bindings with the mpi_f08 Module	40
The call The clared ac	back prototypes when using the Fortran mpi_f08 module are shown below: user-function argument to MPI_Op_create and MPI_Op_create_c should be de- cording to:	41 42 43 44 45
	JTINE MPI_User_function(invec, inoutvec, len, datatype) , INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	46 47 48

```
1
         TYPE(C_PTR), VALUE :: invec, inoutvec
\mathbf{2}
         INTEGER :: len
3
         TYPE(MPI_Datatype) :: datatype
4
     ABSTRACT INTERFACE
5
       SUBROUTINE MPI_User_function_c(invec, inoutvec, len, datatype) !(_c)
6
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
7
         TYPE(C_PTR), VALUE :: invec, inoutvec
8
         INTEGER(KIND=MPI_COUNT_KIND) :: len
9
         TYPE(MPI_Datatype) :: datatype
10
11
         The copy and delete function arguments to MPI_Comm_create_keyval should be de-
12
     clared according to:
13
     ABSTRACT INTERFACE
14
       SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
15
                    attribute_val_in, attribute_val_out, flag, ierror)
16
         TYPE(MPI_Comm) :: oldcomm
17
         INTEGER :: comm_keyval, ierror
18
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
19
                    attribute_val_out
20
         LOGICAL :: flag
21
     ABSTRACT INTERFACE
22
       SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
23
                    attribute_val, extra_state, ierror)
24
         TYPE(MPI_Comm) :: comm
25
         INTEGER :: comm_keyval, ierror
26
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
27
28
         The copy and delete function arguments to MPI_Win_create_keyval should be declared
^{29}
     according to:
30
     ABSTRACT INTERFACE
31
       SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
32
                    attribute_val_in, attribute_val_out, flag, ierror)
33
         TYPE(MPI_Win) :: oldwin
34
         INTEGER :: win_keyval, ierror
35
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
36
                    attribute_val_out
37
         LOGICAL :: flag
38
     ABSTRACT INTERFACE
39
       SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
40
                    extra_state, ierror)
41
         TYPE(MPI_Win) :: win
42
         INTEGER :: win_keyval, ierror
43
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
44
45
         The copy and delete function arguments to MPI_Type_create_keyval should be declared
46
     according to:
47
     ABSTRACT INTERFACE
48
```

```
1
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
                                                                                      2
               attribute_val_in, attribute_val_out, flag, ierror)
                                                                                      3
    TYPE(MPI_Datatype) :: oldtype
    INTEGER :: type_keyval, ierror
                                                                                      4
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                      5
                                                                                      6
               attribute_val_out
    LOGICAL :: flag
                                                                                      7
ABSTRACT INTERFACE
                                                                                      9
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
                                                                                      10
               attribute_val, extra_state, ierror)
                                                                                      11
    TYPE(MPI_Datatype) :: datatype
                                                                                      12
    INTEGER :: type_keyval, ierror
                                                                                      13
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                      14
                                                                                      15
    The handler-function argument to MPI_Comm_create_errhandler should be declared
                                                                                      16
like this:
                                                                                      17
ABSTRACT INTERFACE
                                                                                      18
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
                                                                                      19
    TYPE(MPI_Comm) :: comm
                                                                                      20
    INTEGER :: error_code
                                                                                      21
    The handler-function argument to MPI_Win_create_errhandler should be declared like
                                                                                      22
this:
                                                                                      23
ABSTRACT INTERFACE
                                                                                      24
  SUBROUTINE MPI_Win_errhandler_function(win, error_code)
                                                                                      25
    TYPE(MPI_Win) :: win
                                                                                      26
    INTEGER :: error_code
                                                                                      27
                                                                                      28
    The handler-function argument to MPI_File_create_errhandler should be declared like
                                                                                      29
this:
                                                                                      30
ABSTRACT INTERFACE
                                                                                      31
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
                                                                                      32
    TYPE(MPI_File) :: file
                                                                                      33
    INTEGER :: error_code
                                                                                      34
    The handler-function argument to MPI_Session_create_errhandler should be declared
                                                                                      35
like this:
                                                                                      36
ABSTRACT INTERFACE
                                                                                      37
  SUBROUTINE MPI_Session_errhandler_function(session, error_code)
                                                                                      38
    TYPE(MPI_Session) :: session
                                                                                      39
    INTEGER :: error_code
                                                                                      40
                                                                                      41
    The query, free, and cancel function arguments to MPI_Grequest_start should be de-
                                                                                      42
clared according to:
                                                                                      43
ABSTRACT INTERFACE
                                                                                      44
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
                                                                                      45
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                      46
    TYPE(MPI_Status) :: status
                                                                                      47
    INTEGER :: ierror
                                                                                      48
```

```
1
     ABSTRACT INTERFACE
\mathbf{2}
       SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
3
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
4
         INTEGER :: ierror
5
     ABSTRACT INTERFACE
6
       SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
7
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
8
         LOGICAL :: complete
9
         INTEGER :: ierror
10
11
         The extent and conversion function arguments to MPI_Register_datarep and
12
     MPI_Register_datarep_c should be declared according to:
13
     ABSTRACT INTERFACE
14
       SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
15
                    ierror)
16
         TYPE(MPI_Datatype) :: datatype
17
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
18
         INTEGER :: ierror
19
     ABSTRACT INTERFACE
20
       SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
21
                    filebuf, position, extra_state, ierror)
22
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
23
         TYPE(C_PTR), VALUE :: userbuf, filebuf
24
         TYPE(MPI_Datatype) :: datatype
25
         INTEGER :: count, ierror
26
         INTEGER(KIND=MPI_OFFSET_KIND) :: position
27
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
28
29
     ABSTRACT INTERFACE
30
       SUBROUTINE MPI_Datarep_conversion_function_c(userbuf, datatype, count,
31
                    filebuf, position, extra_state, ierror) !(_c)
32
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
33
         TYPE(C_PTR), VALUE :: userbuf, filebuf
34
         TYPE(MPI_Datatype) :: datatype
35
         INTEGER(KIND=MPI_COUNT_KIND) :: count
36
         INTEGER(KIND=MPI_OFFSET_KIND) :: position
37
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
38
         INTEGER :: ierror
39
40
     Fortran Bindings with mpif.h or the mpi Module
41
42
     With the Fortran mpi module or mpif.h, here are examples of how each of the user-defined
43
     subroutines should be declared.
44
         The user-function argument to MPI_OP_CREATE should be declared like this:
45
     SUBROUTINE USER_FUNCTION (INVEC, INOUTVEC, LEN, DATATYPE)
46
         <type> INVEC(LEN), INOUTVEC(LEN)
47
         INTEGER LEN, DATATYPE
48
```

The copy and delete function arguments to Wint_COMM_CREATE_RETVAL should be	1
declared like these:	2
SUBROUTINE COMM_COPY_ATTR_FUNCTION(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,	3 4
ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)	4 5
INTEGER OLDCOMM, COMM_KEYVAL, IERROR	6
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT	7
LOGICAL FLAG	8
LOGICAL FLAG	9
FYTRA STATE IERROR)	10 11
INTEGER COMM COMM KEYVAL TERBOR	12
INTEGER (KIND=MPI ADDRESS KIND) ATTRIBUTE VAL EXTRA STATE	13
The copy and delete function arguments to MFT_WIN_CREATE_RETVAL should be	$14 \\ 15$
	16
	17
	18
	19
	20
LOGICAL FLAG	21
	22
ΕΥΤΡΑ ΟΤΑΤΕ ΙΕΡΡΟΟΙ	23
TNTECED LITN LITN VEVIAL TEDDOD	24
INTEGED (VIND-MDT ADDRESS VIND) ATTRIDUTE VAL EVTDA STATE	25
	26 27
declared like these:	28
SUBRUUTINE TYPE_CUPY_ATTR_FUNCTION(ULDIYPE, TYPE_KEYVAL, EXTRA_STATE,	29
AIIRIBUIE_VAL_IN, AIIRIBUIE_VAL_UUI, FLAG, IERROR)	30 31
INTEGER OLDTIPE, TIPE_KEIVAL, TERROR	31
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	33
ATTRIBUTE_VAL_UUT	34
LUGICAL FLAG	35
SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,	36 37
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	38
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	39
The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be de-	$40 \\ 41$
	42
	43
	44
<b>0 - - -</b>	45
	46
	47
INTEGER WIN, ERROR_CODE	48

12	The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be de- clared like this:
$\frac{3}{4}$	SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE) INTEGER FILE, ERROR_CODE
5 6 7 8 9	The handler-function argument to MPI_SESSION_CREATE_ERRHANDLER should be declared like this: SUBROUTINE SESSION_ERRHANDLER_FUNCTION(SESSION, ERROR_CODE) INTEGER SESSION, ERROR_CODE
10 11 12 13 14	The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these: SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR) INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE INTEGER STATUS(MPI_STATUS_SIZE), IERROR
15 16 17 18	SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR) INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE INTEGER IERROR
19 20 21 22 23	SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR) INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE LOGICAL COMPLETE INTEGER IERROR
24 25 26 27 28	The extent and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these: SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
29 30 31 32 33 34 35	<pre>SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,</pre>
36 37	A.1.4 Deprecated Prototype Definitions
38 39 40	The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.
41 42	<pre>/* prototypes for user-defined functions */</pre>
43 44 45 46	<pre>typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,</pre>
46 47 48	<pre>typedef int MPI_Delete_function(MPI_Comm comm, int keyval,</pre>

The following are deprecated Fortran user-defined callback subroutine prototypes. The 1  $\mathbf{2}$ deprecated copy and delete function arguments to MPI\_KEYVAL\_CREATE should be de-3 clared like these: SUBROUTINE COPY\_FUNCTION(OLDCOMM, KEYVAL, EXTRA\_STATE, ATTRIBUTE\_VAL\_IN, 4 ATTRIBUTE\_VAL\_OUT, FLAG, IERR) 5INTEGER OLDCOMM, KEYVAL, EXTRA\_STATE, ATTRIBUTE\_VAL\_IN, 6 7 ATTRIBUTE\_VAL\_OUT, IERR 8 LOGICAL FLAG 9 SUBROUTINE DELETE\_FUNCTION(COMM, KEYVAL, ATTRIBUTE\_VAL, EXTRA\_STATE, IERR) 10 INTEGER COMM, KEYVAL, ATTRIBUTE\_VAL, EXTRA\_STATE, IERR 11 1213 A.1.5 String Values 14Default Communicator Names 1516The following default communicator names are defined by MPI. 17 "MPI\_COMM\_WORLD" 18 "MPI\_COMM\_SELF" 19 "MPI\_COMM\_PARENT" 2021**Reserved Data Representations** 22 23The following data representations are supported by MPI.  $^{24}$ "native" 25"internal" 26"external32" 2728 Process Set Names 29Process set name Comment 30 "mpi://" reserved namespace  $^{31}$ "mpi://SELF" mandatory process set name 32 "mpi://WORLD" mandatory process set name 33 34Info Keys 3536 The following info keys are reserved. They are strings. 37 "access\_style" 38 "accumulate\_ops" 39 "accumulate\_ordering" 40 "alloc\_shared\_noncontig" 41 "appnum" 42"arch" 43 "argv" 44"cb\_block\_size" 45"cb\_buffer\_size" 46"cb\_nodes" 47"chunked\_item" 48

- <sup>1</sup> "chunked\_size"
- <sup>2</sup> "chunked"
- <sup>3</sup> "collective\_buffering"
- <sup>4</sup> "command"
- <sup>5</sup> "file"
- <sup>6</sup> "file\_perm"
- <sup>7</sup> "filename"
- <sup>8</sup> "host"
- <sup>9</sup> "io\_node\_list"
- <sup>10</sup> "ip\_address"
- <sup>11</sup> "ip\_port"
- <sup>12</sup> "maxprocs"
- <sup>13</sup> "mpi\_assert\_allow\_overtaking"
- <sup>14</sup> "mpi\_assert\_exact\_length"
- <sup>15</sup> "mpi\_assert\_no\_any\_source"
- <sup>16</sup> "mpi\_assert\_no\_any\_tag"
- <sup>17</sup> "mpi\_hw\_resource\_type"
- <sup>18</sup> "mpi\_initial\_errhandler"
- <sup>19</sup> "mpi\_minimum\_memory\_alignment"
- <sup>20</sup> "mpi\_size"
- <sup>21</sup> "nb\_proc"
- <sup>22</sup> "no\_locks"
- <sup>23</sup> "num\_io\_nodes"
- <sup>24</sup> "path"
- <sup>25</sup> "same\_disp\_unit"
- <sup>26</sup> "same\_size"
- <sup>27</sup> "soft"
- <sup>28</sup> "striping\_factor"
- <sup>29</sup> "striping\_unit"
- <sup>30</sup> "thread\_level"
- <sup>31</sup> "wdir"
- 32
- <sup>33</sup> Info Values
- <sup>35</sup> The following info values are reserved. They are strings.
- <sup>36</sup> "false"
- <sup>37</sup> "mpi\_errors\_abort"
- <sup>38</sup> "mpi\_errors\_are\_fatal"
- <sup>39</sup> "mpi\_errors\_return"
- <sup>40</sup> "mpi\_shared\_memory"
- <sup>41</sup> "MPI\_THREAD\_FUNNELED"
- 42 "MPI\_THREAD\_MULTIPLE"
- 43 "MPI\_THREAD\_SERIALIZED"
- 44 "MPI\_THREAD\_SINGLE"
- 45 "none"
- 46 "random"
- 47 "rar"
- 48 "raw"

"read_mostly"		
"read_once"		
"reverse_sequential"		
"same_op"		
"same_op_no_op"		
"sequential"		
"true"		
"war"		
"waw"		
"write_mostly"		
"write_once"		

## A.2 Summary of the Semantics of all Operation-Related MPI Procedures

A summary of the semantics of all operation-related MPI procedures can be found in [51].

$\frac{1}{2}$	A.3 C Bindings
3	A.3.1 Point-to-Point Communication C Bindings
4 5 6	<pre>int MPI_Bsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
7 8	<pre>int MPI_Bsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
9 10 11	<pre>int MPI_Bsend_init(const void *buf, int count, MPI_Datatype datatype,</pre>
12 13 14	<pre>int MPI_Bsend_init_c(const void *buf, MPI_Count count,</pre>
15 16	<pre>int MPI_Buffer_attach(void *buffer, int size)</pre>
17	<pre>int MPI_Buffer_attach_c(void *buffer, MPI_Count size)</pre>
18 19	<pre>int MPI_Buffer_detach(void *buffer_addr, int *size)</pre>
20	<pre>int MPI_Buffer_detach_c(void *buffer_addr, MPI_Count *size)</pre>
21 22	<pre>int MPI_Cancel(MPI_Request *request)</pre>
23 24	<pre>int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype,</pre>
25 26 27	<pre>int MPI_Get_count_c(const MPI_Status *status, MPI_Datatype datatype,</pre>
28 29 30	<pre>int MPI_Ibsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>
31 32	<pre>int MPI_Ibsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
33 34 35	<pre>int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Message *message, MPI_Status *status)</pre>
36 37	<pre>int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,</pre>
38 39 40	<pre>int MPI_Imrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>
41 42	<pre>int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,</pre>
43 44 45	<pre>int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,</pre>
46 47 48	<pre>int MPI_Irecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>

int	<pre>MPI_Irsend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	$\frac{1}{2}$
int	<pre>MPI_Irsend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	3 4 5
int	<pre>MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	6 7
int	<pre>MPI_Isend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	8 9 10
int	<pre>MPI_Isendrecv(const void *sendbuf, int sendcount,</pre>	11 12 13 14 15
int	<pre>MPI_Isendrecv_c(const void *sendbuf, MPI_Count sendcount,</pre>	16 17 18 19
int	<pre>MPI_Isendrecv_replace(void *buf, int count, MPI_Datatype datatype,</pre>	20 21 22 23
int	<pre>MPI_Isendrecv_replace_c(void *buf, MPI_Count count,</pre>	24 25 26
int	<pre>MPI_Issend(const void *buf, int count, MPI_Datatype datatype, int dest,</pre>	27 28 29
int	<pre>MPI_Issend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	30 31
int	<pre>MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message, MPI_Status *status)</pre>	32 33 34
int	<pre>MPI_Mrecv(void *buf, int count, MPI_Datatype datatype,</pre>	35 36
int	<pre>MPI_Mrecv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	37 38 39
int	<pre>MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)</pre>	40
int	<pre>MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,</pre>	41 42 43
int	<pre>MPI_Recv_c(void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	44 45
int	<pre>MPI_Recv_init(void *buf, int count, MPI_Datatype datatype, int source,</pre>	46 47 48

1 int MPI\_Recv\_init\_c(void \*buf, MPI\_Count count, MPI\_Datatype datatype,  $\mathbf{2}$ int source, int tag, MPI\_Comm comm, MPI\_Request \*request) 3 int MPI\_Request\_free(MPI\_Request \*request) 4  $\mathbf{5}$ int MPI\_Request\_get\_status(MPI\_Request request, int \*flag, 6 MPI\_Status \*status) 7 int MPI\_Rsend(const void \*buf, int count, MPI\_Datatype datatype, int dest, 8 int tag, MPI\_Comm comm) 9 10int MPI\_Rsend\_c(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 11 int dest, int tag, MPI\_Comm comm) 12int MPI\_Rsend\_init(const void \*buf, int count, MPI\_Datatype datatype, 13 int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 1415int MPI\_Rsend\_init\_c(const void \*buf, MPI\_Count count, 16MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm, 17MPI\_Request \*request) 18 int MPI\_Send(const void \*buf, int count, MPI\_Datatype datatype, int dest, 19int tag, MPI\_Comm comm) 2021int MPI\_Send\_c(const void \*buf, MPI\_Count count, MPI\_Datatype datatype, 22int dest, int tag, MPI\_Comm comm) 23int MPI\_Send\_init(const void \*buf, int count, MPI\_Datatype datatype,  $^{24}$ int dest, int tag, MPI\_Comm comm, MPI\_Request \*request) 2526int MPI\_Send\_init\_c(const void \*buf, MPI\_Count count, 27MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm, 28MPI\_Request \*request) 29int MPI\_Sendrecv(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 30 int dest, int sendtag, void \*recvbuf, int recvcount,  $^{31}$ MPI\_Datatype recvtype, int source, int recvtag, MPI\_Comm comm, 32 MPI\_Status \*status) 33 34int MPI\_Sendrecv\_c(const void \*sendbuf, MPI\_Count sendcount, 35 MPI\_Datatype sendtype, int dest, int sendtag, void \*recvbuf, 36 MPI\_Count recvcount, MPI\_Datatype recvtype, int source, 37 int recvtag, MPI\_Comm comm, MPI\_Status \*status) 38 int MPI\_Sendrecv\_replace(void \*buf, int count, MPI\_Datatype datatype, 39int dest, int sendtag, int source, int recvtag, MPI\_Comm comm, 4041 MPI\_Status \*status) 42int MPI\_Sendrecv\_replace\_c(void \*buf, MPI\_Count count, 43 MPI\_Datatype datatype, int dest, int sendtag, int source, 44 int recvtag, MPI\_Comm comm, MPI\_Status \*status) 4546int MPI\_Ssend(const void \*buf, int count, MPI\_Datatype datatype, int dest, 47int tag, MPI\_Comm comm) 48

int	<pre>MPI_Ssend_c(const void *buf, MPI_Count count, MPI_Datatype datatype,</pre>	1 2
int	<pre>MPI_Ssend_init(const void *buf, int count, MPI_Datatype datatype,</pre>	3 4 5
int	<pre>MPI_Ssend_init_c(const void *buf, MPI_Count count,</pre>	6 7 8
int	MPI_Start(MPI_Request *request)	9 10
int	<pre>MPI_Startall(int count, MPI_Request array_of_requests[])</pre>	11
int	MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)	12 13
	MPI_Test_cancelled(const MPI_Status *status, int *flag)	14
	<pre>MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_of_statuses[])</pre>	15 16 17
int	<pre>MPI_Testany(int count, MPI_Request array_of_requests[], int *index,</pre>	18 19 20
int	<pre>MPI_Testsome(int incount, MPI_Request array_of_requests[],</pre>	21 22 23
int	MPI_Wait(MPI_Request *request, MPI_Status *status)	24
int	<pre>MPI_Waitall(int count, MPI_Request array_of_requests[],</pre>	25 26 27
int	<pre>MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)</pre>	28 29 30
int	<pre>MPI_Waitsome(int incount, MPI_Request array_of_requests[],</pre>	31 32 33 34
A.3.	.2 Partitioned Communication C Bindings	35 36
	MPI_Parrived(MPI_Request request, int partition, int *flag)	37
	MPI_Pready(int partition, MPI_Request request)	38
		39 40
int	<pre>MPI_Pready_list(int length, const int array_of_partitions[],</pre>	41 42
int	<pre>MPI_Pready_range(int partition_low, int partition_high,</pre>	43 44
int	<pre>MPI_Precv_init(void *buf, int partitions, MPI_Count count,</pre>	45 46 47 48

1 2 3 4	int	<pre>MPI_Psend_init(const void *buf, int partitions, MPI_Count count,</pre>
5 6	A.3.	3 Datatypes C Bindings
7	MPI_	_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
8 9	MPI_	_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)
10	int	<pre>MPI_Get_address(const void *location, MPI_Aint *address)</pre>
11 12 13	int	<pre>MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype,</pre>
14 15 16	int	<pre>MPI_Get_elements_c(const MPI_Status *status, MPI_Datatype datatype,</pre>
17 18	int	<pre>MPI_Get_elements_x(const MPI_Status *status, MPI_Datatype datatype,</pre>
19 20 21	int	<pre>MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)</pre>
22 23 24	int	<pre>MPI_Pack_c(const void *inbuf, MPI_Count incount, MPI_Datatype datatype, void *outbuf, MPI_Count outsize, MPI_Count *position, MPI_Comm comm)</pre>
25 26 27 28	int	<pre>MPI_Pack_external(const char datarep[], const void *inbuf, int incount,</pre>
29 30 31	int	<pre>MPI_Pack_external_c(const char datarep[], const void *inbuf,</pre>
32 33 34	int	<pre>MPI_Pack_external_size(const char datarep[], int incount,</pre>
35 36	int	<pre>MPI_Pack_external_size_c(const char datarep[], MPI_Count incount,</pre>
37 38 39	int	<pre>MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,</pre>
40 41 42	int	<pre>MPI_Pack_size_c(MPI_Count incount, MPI_Datatype datatype,</pre>
42	int	MPI_Type_commit(MPI_Datatype *datatype)
44 45 46	int	<pre>MPI_Type_contiguous(int count, MPI_Datatype oldtype,</pre>
47 48	int	<pre>MPI_Type_contiguous_c(MPI_Count count, MPI_Datatype oldtype,</pre>

$\operatorname{int}$	<pre>MPI_Type_create_darray(int size, int rank, int ndims,</pre>	1
	<pre>const int array_of_gsizes[], const int array_of_distribs[],</pre>	2
	<pre>const int array_of_dargs[], const int array_of_psizes[],</pre>	3
	<pre>int order, MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	4
int	MPI_Type_create_darray_c(int size, int rank, int ndims,	5
THC	const MPI_Count array_of_gsizes[],	6
		7
	<pre>const int array_of_distribs[], const int array_of_dargs[], const int array_of_gizes[],</pre>	8
	<pre>const int array_of_psizes[], int order, MPI_Datatype oldtype, MDI_Datatype in order.</pre>	9
	MPI_Datatype *newtype)	10
int	<pre>MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],</pre>	11
	<pre>const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,</pre>	12
	MPI_Datatype *newtype)	13
		14
int	MPI_Type_create_hindexed_block(int count, int blocklength,	15
	<pre>const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,</pre>	16
	MPI_Datatype *newtype)	17
int	MPI_Type_create_hindexed_block_c(MPI_Count count,	18
	MPI_Count blocklength,	19
	<pre>const MPI_Count array_of_displacements[],</pre>	20
	MPI_Datatype oldtype, MPI_Datatype *newtype)	21
		22
int	MPI_Type_create_hindexed_c(MPI_Count count,	23
	<pre>const MPI_Count array_of_blocklengths[],</pre>	$^{24}$
	<pre>const MPI_Count array_of_displacements[],</pre>	25
	MPI_Datatype oldtype, MPI_Datatype *newtype)	26
int	MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,	27
THC	MPI_Datatype oldtype, MPI_Datatype *newtype)	28
	MI_Datatype Oldtype, MI_Datatype *newtype)	29
int	<pre>MPI_Type_create_hvector_c(MPI_Count count, MPI_Count blocklength,</pre>	30
	MPI_Count stride, MPI_Datatype oldtype, MPI_Datatype *newtype)	31
÷	MDT True success indexed block(int sound int block] anoth	32
int	MPI_Type_create_indexed_block(int count, int blocklength,	33
	<pre>const int array_of_displacements[], MPI_Datatype oldtype,</pre>	34
	MPI_Datatype *newtype)	35
int	MPI_Type_create_indexed_block_c(MPI_Count count, MPI_Count blocklength,	36
	<pre>const MPI_Count array_of_displacements[],</pre>	37
	MPI_Datatype oldtype, MPI_Datatype *newtype)	38
		39
int	MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,	40
	MPI_Aint extent, MPI_Datatype *newtype)	41
int	MPI_Type_create_resized_c(MPI_Datatype oldtype, MPI_Count lb,	42
	MPI_Count extent, MPI_Datatype *newtype)	43
		44
int	<pre>MPI_Type_create_struct(int count, const int array_of_blocklengths[],</pre>	45
	<pre>const MPI_Aint array_of_displacements[],</pre>	46
	<pre>const MPI_Datatype array_of_types[], MPI_Datatype *newtype)</pre>	47
in+	MPI_Type_create_struct_c(MPI_Count count,	48
	In T_Type_orouto_burdeb_o(In T_oound County,	

```
1
                   const MPI_Count array_of_blocklengths[],
2
                   const MPI_Count array_of_displacements[],
3
                   const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
4
     int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
5
                   const int array_of_subsizes[], const int array_of_starts[],
6
                   int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
7
8
     int MPI_Type_create_subarray_c(int ndims, const MPI_Count array_of_sizes[],
9
                   const MPI_Count array_of_subsizes[],
10
                   const MPI_Count array_of_starts[], int order,
11
                   MPI_Datatype oldtype, MPI_Datatype *newtype)
12
     int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
13
14
     int MPI_Type_free(MPI_Datatype *datatype)
15
     int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
16
                   int max_addresses, int max_datatypes, int array_of_integers[],
17
                   MPI_Aint array_of_addresses[],
18
                   MPI_Datatype array_of_datatypes[])
19
20
     int MPI_Type_get_contents_c(MPI_Datatype datatype, MPI_Count max_integers,
21
                   MPI_Count max_addresses, MPI_Count max_large_counts,
22
                   MPI_Count max_datatypes, int array_of_integers[],
23
                  MPI_Aint array_of_addresses[],
24
                   MPI_Count array_of_large_counts[],
25
                   MPI_Datatype array_of_datatypes[])
26
     int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
27
                   int *num_addresses, int *num_datatypes, int *combiner)
28
29
     int MPI_Type_get_envelope_c(MPI_Datatype datatype, MPI_Count *num_integers,
30
                   MPI_Count *num_addresses, MPI_Count *num_large_counts,
^{31}
                   MPI_Count *num_datatypes, int *combiner)
32
     int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
33
                  MPI_Aint *extent)
34
35
     int MPI_Type_get_extent_c(MPI_Datatype datatype, MPI_Count *lb,
36
                   MPI_Count *extent)
37
     int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *1b,
38
                   MPI_Count *extent)
39
40
     int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
41
                   MPI_Aint *true_extent)
42
     int MPI_Type_get_true_extent_c(MPI_Datatype datatype, MPI_Count *true_lb,
43
                  MPI_Count *true_extent)
44
45
     int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
46
                   MPI_Count *true_extent)
47
48
     int MPI_Type_indexed(int count, const int array_of_blocklengths[],
```

	<pre>const int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	1 2
int	<pre>MPI_Type_indexed_c(MPI_Count count,</pre>	3 4 5
	<pre>const MPI_Count array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)</pre>	6 7
int	<pre>MPI_Type_size(MPI_Datatype datatype, int *size)</pre>	8
int	MPI_Type_size_c(MPI_Datatype datatype, MPI_Count *size)	9 10
int	MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)	11
int	<pre>MPI_Type_vector(int count, int blocklength, int stride,</pre>	12 13 14
int	<pre>MPI_Type_vector_c(MPI_Count count, MPI_Count blocklength,</pre>	15 16 17
int	<pre>MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,</pre>	18 19
int	<pre>MPI_Unpack_c(const void *inbuf, MPI_Count insize, MPI_Count *position, void *outbuf, MPI_Count outcount, MPI_Datatype datatype, MPI_Comm comm)</pre>	20 21 22 23
int	<pre>MPI_Unpack_external(const char datarep[], const void *inbuf,</pre>	24 25 26
int	<pre>MPI_Unpack_external_c(const char datarep[], const void *inbuf,</pre>	27 28 29 30
A.3.	.4 Collective Communication C Bindings	31 32
int	<pre>MPI_Allgather(const void *sendbuf, int sendcount,</pre>	33 34 35 36
int	<pre>MPI_Allgather_c(const void *sendbuf, MPI_Count sendcount,</pre>	37 38 39
int	<pre>MPI_Allgather_init(const void *sendbuf, int sendcount,</pre>	40 41 42 43
int	<pre>MPI_Request *request) MPI_Allgather_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	44 45 46 47 48

1MPI\_Request \*request)  $\mathbf{2}$ int MPI\_Allgatherv(const void \*sendbuf, int sendcount, 3 MPI\_Datatype sendtype, void \*recvbuf, const int recvcounts[], 4 const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm) 56 int MPI\_Allgatherv\_c(const void \*sendbuf, MPI\_Count sendcount, 7 MPI\_Datatype sendtype, void \*recvbuf, 8 const MPI\_Count recvcounts[], const MPI\_Aint displs[], 9 MPI\_Datatype recvtype, MPI\_Comm comm) 10 int MPI\_Allgatherv\_init(const void \*sendbuf, int sendcount, 11 MPI\_Datatype sendtype, void \*recvbuf, const int recvcounts[], 12const int displs[], MPI\_Datatype recvtype, MPI\_Comm comm, 13 MPI\_Info info, MPI\_Request \*request) 1415int MPI\_Allgatherv\_init\_c(const void \*sendbuf, MPI\_Count sendcount, 16MPI\_Datatype sendtype, void \*recvbuf, 17const MPI\_Count recvcounts[], const MPI\_Aint displs[], 18 MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Info info, 19MPI\_Request \*request) 20int MPI\_Allreduce(const void \*sendbuf, void \*recvbuf, int count, 21MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm) 22 23int MPI\_Allreduce\_c(const void \*sendbuf, void \*recvbuf, MPI\_Count count, 24MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm) 25int MPI\_Allreduce\_init(const void \*sendbuf, void \*recvbuf, int count, 26MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm, 27MPI\_Info info, MPI\_Request \*request) 2829 int MPI\_Allreduce\_init\_c(const void \*sendbuf, void \*recvbuf, 30 MPI\_Count count, MPI\_Datatype datatype, MPI\_Op op, 31MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request) 32 int MPI\_Alltoall(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype, 33 void \*recvbuf, int recvcount, MPI\_Datatype recvtype, 34 MPI\_Comm comm) 35 36 int MPI\_Alltoall\_c(const void \*sendbuf, MPI\_Count sendcount, 37 MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 38 MPI\_Datatype recvtype, MPI\_Comm comm) 39 int MPI\_Alltoall\_init(const void \*sendbuf, int sendcount, 40MPI\_Datatype sendtype, void \*recvbuf, int recvcount, 41 42MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Info info, MPI\_Request \*request) 43 44int MPI\_Alltoall\_init\_c(const void \*sendbuf, MPI\_Count sendcount, 45 MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount, 46MPI\_Datatype recvtype, MPI\_Comm comm, MPI\_Info info, 47 MPI\_Request \*request) 48

int	<pre>MPI_Alltoallv(const void *sendbuf, const int sendcounts[],</pre>	1 2 3 4
int	<pre>MPI_Alltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	5 6 7 8 9
int	<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,</pre>	11 12 13 14 15 16
int	<pre>const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint rdispls[], MPI_Datatype recvtype, MPI_Comm_commMPI_Info_infoMPI_Poquest_trequest)</pre>	10 17 18 19 20 21
int	<pre>void *recvbuf, const int recvcounts[], const int rdispls[],</pre>	22 23 24 25 26
int	<pre>const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MDL Comm_comm)</pre>	27 28 29 30 31
int	<pre>MPI_Alltoallw_init(const void *sendbur, const int sendcounts[],</pre>	32 33 34 35 36 37
int	<pre>MPI_Alltoallw_init_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	38 39 40 41 42
int	MPI_Barrier(MPI_Comm comm)	43 44
int	<pre>MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	45
int	MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,	46 47 48

```
1
     int MPI_Bcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
\mathbf{2}
                   int root, MPI_Comm comm)
3
     int MPI_Bcast_init(void *buffer, int count, MPI_Datatype datatype,
4
                   int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
5
6
     int MPI_Bcast_init_c(void *buffer, MPI_Count count, MPI_Datatype datatype,
7
                   int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
8
     int MPI_Exscan(const void *sendbuf, void *recvbuf, int count,
9
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
10
11
     int MPI_Exscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,
12
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
13
     int MPI_Exscan_init(const void *sendbuf, void *recvbuf, int count,
14
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
15
                   MPI_Info info, MPI_Request *request)
16
17
     int MPI_Exscan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,
18
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
19
                   MPI_Info info, MPI_Request *request)
20
     int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
21
                   void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
22
                   MPI_Comm comm)
23
24
     int MPI_Gather_c(const void *sendbuf, MPI_Count sendcount,
25
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
26
                   MPI_Datatype recvtype, int root, MPI_Comm comm)
27
     int MPI_Gather_init(const void *sendbuf, int sendcount,
28
                  MPI_Datatype sendtype, void *recvbuf, int recvcount,
29
                   MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
30
                  MPI_Request *request)
^{31}
32
     int MPI_Gather_init_c(const void *sendbuf, MPI_Count sendcount,
33
                   MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
34
                   MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
35
                  MPI_Request *request)
36
     int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
37
                   void *recvbuf, const int recvcounts[], const int displs[],
38
                   MPI_Datatype recvtype, int root, MPI_Comm comm)
39
40
     int MPI_Gatherv_c(const void *sendbuf, MPI_Count sendcount,
41
                   MPI_Datatype sendtype, void *recvbuf,
42
                   const MPI_Count recvcounts[], const MPI_Aint displs[],
43
                   MPI_Datatype recvtype, int root, MPI_Comm comm)
44
     int MPI_Gatherv_init(const void *sendbuf, int sendcount,
45
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
46
47
                   const int displs[], MPI_Datatype recvtype, int root,
48
                   MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

int	<pre>MPI_Gatherv_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	1
	MPI_Datatype sendtype, void *recvbuf,	2
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>	3
	MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,	4
	MPI_Request *request)	$\frac{5}{6}$
int	<pre>IPI_Iallgather(const void *sendbuf, int sendcount,</pre>	7
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	8
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	9
int	<pre>/PI_Iallgather_c(const void *sendbuf, MPI_Count sendcount,</pre>	10
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	11
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	12
int	<pre>Interv(const void *sendbuf, int sendcount,</pre>	13 14
	MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],	15
	const int displs[], MPI_Datatype recvtype, MPI_Comm comm,	16
	MPI_Request *request)	17
in+	<pre>MPI_Iallgatherv_c(const void *sendbuf, MPI_Count sendcount,</pre>	18
1110	MPI_Datatype sendtype, void *recvbuf,	19
	const MPI_Count recvcounts[], const MPI_Aint displs[],	20
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	21
		22
int	<pre>MPI_Iallreduce(const void *sendbuf, void *recvbuf, int count,</pre>	23
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	24
	MPI_Request *request)	25
int	<pre>MPI_Iallreduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	26
	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	27
	MPI_Request *request)	28
int	<pre>/PI_Ialltoall(const void *sendbuf, int sendcount,</pre>	29 30
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	31
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)	32
·		33
int	<pre>MPI_Ialltoall_c(const void *sendbuf, MPI_Count sendcount,</pre>	34
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	35
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	36
int	<pre>PI_Ialltoallv(const void *sendbuf, const int sendcounts[],</pre>	37
	<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>	38
	<pre>const int recvcounts[], const int rdispls[],</pre>	39
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	40
int	<pre>MPI_Ialltoallv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	41
	<pre>const MPI_Aint sdispls[], MPI_Datatype sendtype,</pre>	42
	void *recvbuf, const MPI_Count recvcounts[],	43
	<pre>const MPI_Aint rdispls[], MPI_Datatype recvtype,</pre>	44
	MPI_Comm comm, MPI_Request *request)	45
		46
TUL	Image: API_Ialltoallw(const void *sendbuf, const int sendcounts[],         const int sdispla[]         const int sdispla[]	47
	<pre>const int sdispls[], const MPI_Datatype sendtypes[],</pre>	48

1 2 3 4		<pre>void *recvbuf, const int recvcounts[], const int rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)</pre>
4 5 6 7 8 9	int MPI	<pre>[_Ialltoallw_c(const void *sendbuf, const MPI_Count sendcounts[], const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Request *request)</pre>
10	int MPI	_Ibarrier(MPI_Comm comm, MPI_Request *request)
11 12 13	int MPI	<pre>_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)</pre>
14 15	int MPI	<pre>_Ibcast_c(void *buffer, MPI_Count count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)</pre>
16 17 18 19	int MPI	<pre>[_lexscan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>
20 21 22 23	int MPI	<pre>[_lexscan_c(const void *sendbuf, void *recvbuf, MPI_Count count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>
23 24 25 26	int MPI	<pre>_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
27 28 29 30 31	int MPI	<pre>[_Igather_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
32 33 34 35	int MPI	<pre>[_Igatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
36 37 38 39 40 41	int MPI	<pre>[_Igatherv_c(const void *sendbuf, MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf, const MPI_Count recvcounts[], const MPI_Aint displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)</pre>
42 43 44	int MPI	<pre>_Ireduce(const void *sendbuf, void *recvbuf, int count,</pre>
45 46 47 48	int MPI	<pre>_Ireduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>

int	<pre>MPI_Ireduce_scatter(const void *sendbuf, void *recvbuf,</pre>	1 2 3	
	MPI_Comm comm, MPI_Request *request)	4	
int	<pre>MPI_Ireduce_scatter_block(const void *sendbuf, void *recvbuf,</pre>	5 6	
	MPI_Comm comm, MPI_Request *request)	7	
int	<pre>MPI_Ireduce_scatter_block_c(const void *sendbuf, void *recvbuf,</pre>	8	
	<pre>MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)</pre>	9 10	
int	<pre>MPI_Ireduce_scatter_c(const void *sendbuf, void *recvbuf,</pre>	11	
1110	const MPI_Count recvcounts[], MPI_Datatype datatype,	12	
	MPI_Op op, MPI_Comm comm, MPI_Request *request)	13 14	
int	MPI_Iscan(const void *sendbuf, void *recvbuf, int count,	15	
1110	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,		
	MPI_Request *request)	17	
int	<pre>MPI_Iscan_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	18 19	
1110	MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	20	
	MPI_Request *request)	20	
int	MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,	22	
IIIC	void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,	23	
	MPI_Comm comm, MPI_Request *request)	24	
÷	MPI_Iscatter_c(const void *sendbuf, MPI_Count sendcount,	25	
THE	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	26 27	
	MPI_Datatype recvtype, int root, MPI_Comm comm,	27	
	MPI_Request *request)	20	
int	<pre>MPI_Iscatterv(const void *sendbuf, const int sendcounts[],</pre>	30	
TUC	const int displs[], MPI_Datatype sendtype, void *recvbuf,	31	
	int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,	32	
	MPI_Request *request)	33	
		34	
int	MPI_Iscatterv_c(const void *sendbuf, const MPI_Count sendcounts[],	35	
	<pre>const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount, MPI_Datatype recvtype, int root,</pre>	36 37	
	MPI_Comm comm, MPI_Request *request)	38	
int	<pre>MPI_Op_commutative(MPI_Op op, int *commute)</pre>	39	
		40	
int	<pre>MPI_Op_create(MPI_User_function *user_fn, int commute, MPI_Op *op)</pre>	41 42	
int	<pre>MPI_Op_create_c(MPI_User_function_c *user_fn, int commute, MPI_Op *op)</pre>	42	
int	MPI_Op_free(MPI_Op *op)	44	
int	MPI_Reduce(const void *sendbuf, void *recvbuf, int count,	45 46	
	<pre>MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)</pre>	40	
int	MPI_Reduce_c(const void *sendbuf, void *recvbuf, MPI_Count count,	48	

1		MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
2	int MPT Redu	<pre>init(const void *sendbuf, void *recvbuf, int count,</pre>
3 4		MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
5		MPI_Info info, MPI_Request *request)
6	int MPI_Redu	<pre>nce_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>
7	_	MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
8		<pre>MPI_Info info, MPI_Request *request)</pre>
9 10	int MPI_Redu	<pre>nce_local(const void *inbuf, void *inoutbuf, int count,</pre>
11		MPI_Datatype datatype, MPI_Op op)
12	int MPI Redu	<pre>nce_local_c(const void *inbuf, void *inoutbuf, MPI_Count count,</pre>
13		MPI_Datatype datatype, MPI_Op op)
14	int MPT Redu	<pre>nce_scatter(const void *sendbuf, void *recvbuf,</pre>
15 16	int in i_nout	const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
17		MPI_Comm comm)
18	int MPI Redu	<pre>nce_scatter_block(const void *sendbuf, void *recvbuf,</pre>
19		int recvcount, MPI_Datatype datatype, MPI_Op op,
20		MPI_Comm comm)
21 22	int MPI_Redu	<pre>nce_scatter_block_c(const void *sendbuf, void *recvbuf,</pre>
23	_	MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,
24		MPI_Comm comm)
25	int MPI_Redu	<pre>nce_scatter_block_init(const void *sendbuf, void *recvbuf,</pre>
26 27		<pre>int recvcount, MPI_Datatype datatype, MPI_Op op,</pre>
28		<pre>MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
29	int MPI_Redu	<pre>ace_scatter_block_init_c(const void *sendbuf, void *recvbuf,</pre>
30		<pre>MPI_Count recvcount, MPI_Datatype datatype, MPI_Op op,</pre>
31		<pre>MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
32 33	int MPI_Redu	<pre>ace_scatter_c(const void *sendbuf, void *recvbuf,</pre>
34		<pre>const MPI_Count recvcounts[], MPI_Datatype datatype,</pre>
35		MPI_Op op, MPI_Comm comm)
36	int MPI_Redu	<pre>ice_scatter_init(const void *sendbuf, void *recvbuf,</pre>
37		const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
38 39		<pre>MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
40	int MPI_Redu	<pre>ice_scatter_init_c(const void *sendbuf, void *recvbuf,</pre>
41		const MPI_Count recvcounts[], MPI_Datatype datatype,
42		<pre>MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>
43	int MPI_Scan	(const void *sendbuf, void *recvbuf, int count,
44 45		MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
46	int MPI_Scan	n_c(const void *sendbuf, void *recvbuf, MPI_Count count,
47		MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
48	int MPI_Scan	n_init(const void *sendbuf, void *recvbuf, int count,

<pre>MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request)</pre>	1 2
	3
<pre>int MPI_Scan_init_c(const void *sendbuf, void *recvbuf, MPI_Count count,</pre>	4
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,	5
MPI_Info info, MPI_Request *request)	6
<pre>int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype)</pre>	7
void *recvbuf, int recvcount, MPI_Datatype recvtype, int roo	
MPI_Comm comm)	9
	10
<pre>int MPI_Scatter_c(const void *sendbuf, MPI_Count sendcount,</pre>	11
<pre>MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	12
MPI_Datatype recvtype, int root, MPI_Comm comm)	13
	14
<pre>int MPI_Scatter_init(const void *sendbuf, int sendcount,</pre>	15
MPI_Datatype sendtype, void *recvbuf, int recvcount,	
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info inf	0, <sup>10</sup> 17
MPI_Request *request)	18
<pre>int MPI_Scatter_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	
MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	19
MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info inf	20
	0, 21
MPI_Request *request)	22
<pre>int MPI_Scatterv(const void *sendbuf, const int sendcounts[],</pre>	23
<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	24
int recvcount, MPI_Datatype recvtype, int root, MPI_Comm com	m) $^{25}$
	26
<pre>int MPI_Scatterv_c(const void *sendbuf, const MPI_Count sendcounts[],</pre>	27
<pre>const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbu</pre>	f, <sub>28</sub>
<pre>MPI_Count recvcount, MPI_Datatype recvtype, int root,</pre>	29
MPI_Comm comm)	30
int NDT Grottern init(court will tree live) or the court of	31
<pre>int MPI_Scatterv_init(const void *sendbuf, const int sendcounts[],</pre>	32
<pre>const int displs[], MPI_Datatype sendtype, void *recvbuf,</pre>	33
<pre>int recvcount, MPI_Datatype recvtype, int root, MPI_Comm com</pre>	<b>m</b> , 34
MPI_Info info, MPI_Request *request)	35
<pre>int MPI_Scatterv_init_c(const void *sendbuf, const MPI_Count sendcounts[]</pre>	
const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbu	
MPI_Count recvcount, MPI_Datatype recvtype, int root,	38
MPI_Comm comm, MPI_Info info, MPI_Request *request)	39
in r_comm comm, in r_into into, in r_nequest *request)	
	40
A.3.5 Groups, Contexts, Communicators, and Caching C Bindings	41
	42
<pre>int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)</pre>	43
<pre>int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)</pre>	44
	45
<pre>int MPI_Comm_create_from_group(MPI_Group group, const char *stringtag,</pre>	46
<pre>MPI_Info info, MPI_Errhandler errhandler, MPI_Comm *newcomm)</pre>	47
	48

1 2	int	<pre>MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,</pre>
3	int	<pre>MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,</pre>
4 5		MPI_Comm_delete_attr_function *comm_delete_attr_fn,
6		<pre>int *comm_keyval, void *extra_state)</pre>
7 8	int	<pre>MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)</pre>
9	int	MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
10 11	int	<pre>MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
12 13	int	MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
14 15	int	MPI_Comm_free(MPI_Comm *comm)
16	int	MPI_Comm_free_keyval(int *comm_keyval)
17	int	MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
18 19		int *flag)
20	int	MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
21 22	int	<pre>MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)</pre>
23	int	MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
24 25	int	MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
26	int	MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,
27 28		MPI_Comm *newcomm, MPI_Request *request)
29	int	MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval,
30		<pre>void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag)</pre>
31		Ŭ
32 33	int	<pre>MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval,</pre>
34		
35	int	MPI_Comm_rank(MPI_Comm comm, int *rank)
36 37	int	<pre>MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)</pre>
38	int	<pre>MPI_Comm_remote_size(MPI_Comm comm, int *size)</pre>
39	int	<pre>MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)</pre>
40 41	int	MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
42	int	MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
43 44	int	MPI_Comm_size(MPI_Comm comm, int *size)
45	int	MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
46 47	jnt	<pre>MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,</pre>
48		MPI_Info info, MPI_Comm *newcomm)

<pre>int MPI_Comm_test_inter(MPI_Comm comm, int *flag)</pre>	1
<pre>int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)</pre>	2 3
<pre>int MPI_Group_difference(MPI_Group group1, MPI_Group group2,</pre>	4 5
<pre>int MPI_Group_excl(MPI_Group group, int n, const int ranks[],</pre>	6 7
int MPI_Group_free(MPI_Group *group)	8 9
<pre>int MPI_Group_from_session_pset(MPI_Session session, const char *pset_name,</pre>	10 11 12
<pre>int MPI_Group_incl(MPI_Group group, int n, const int ranks[],</pre>	13 14
<pre>int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,</pre>	15 16 17
<pre>int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],</pre>	18 19
<pre>int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],</pre>	20 21 22
<pre>int MPI_Group_rank(MPI_Group group, int *rank)</pre>	23 24
<pre>int MPI_Group_size(MPI_Group group, int *size)</pre>	25
<pre>int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],</pre>	26 27 28
<pre>int MPI_Group_union(MPI_Group group1, MPI_Group group2,</pre>	29 30
<pre>int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,</pre>	31 32 33 34
<pre>int MPI_Intercomm_create_from_groups(MPI_Group local_group,</pre>	35 36 37 38
<pre>int MPI_Intercomm_merge(MPI_Comm intercomm, int high,</pre>	39 40 41
<pre>int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn,</pre>	42 43 44
<pre>int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)</pre>	45 46
<pre>int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval,</pre>	47 48

1void \*extra\_state, void \*attribute\_val\_in,  $\mathbf{2}$ void \*attribute\_val\_out, int \*flag) 3 int MPI\_Type\_free\_keyval(int \*type\_keyval) 4  $\mathbf{5}$ int MPI\_Type\_get\_attr(MPI\_Datatype datatype, int type\_keyval, 6 void \*attribute\_val, int \*flag) 7 int MPI\_Type\_get\_name(MPI\_Datatype datatype, char \*type\_name, 8 int \*resultlen) 9 10int MPI\_TYPE\_NULL\_COPY\_FN(MPI\_Datatype oldtype, int type\_keyval, 11 void \*extra\_state, void \*attribute\_val\_in, 12void \*attribute\_val\_out, int \*flag) 13int MPI\_TYPE\_NULL\_DELETE\_FN(MPI\_Datatype datatype, int type\_keyval, 14void \*attribute\_val, void \*extra\_state) 1516int MPI\_Type\_set\_attr(MPI\_Datatype datatype, int type\_keyval, 17void \*attribute\_val) 18 int MPI\_Type\_set\_name(MPI\_Datatype datatype, const char \*type\_name) 1920int MPI\_Win\_create\_keyval(MPI\_Win\_copy\_attr\_function \*win\_copy\_attr\_fn, 21MPI\_Win\_delete\_attr\_function \*win\_delete\_attr\_fn, 22int \*win\_keyval, void \*extra\_state) 23int MPI\_Win\_delete\_attr(MPI\_Win win, int win\_keyval)  $^{24}$ 25int MPI\_WIN\_DUP\_FN(MPI\_Win oldwin, int win\_keyval, void \*extra\_state, 26void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag) 27int MPI\_Win\_free\_keyval(int \*win\_keyval) 2829int MPI\_Win\_get\_attr(MPI\_Win win, int win\_keyval, void \*attribute\_val, 30 int \*flag)  $^{31}$ int MPI\_Win\_get\_name(MPI\_Win win, char \*win\_name, int \*resultlen) 3233 int MPI\_WIN\_NULL\_COPY\_FN(MPI\_Win oldwin, int win\_keyval, void \*extra\_state, 34 void \*attribute\_val\_in, void \*attribute\_val\_out, int \*flag) 3536 int MPI\_WIN\_NULL\_DELETE\_FN(MPI\_Win win, int win\_keyval, 37 void \*attribute\_val, void \*extra\_state) 38int MPI\_Win\_set\_attr(MPI\_Win win, int win\_keyval, void \*attribute\_val) 39 40int MPI\_Win\_set\_name(MPI\_Win win, const char \*win\_name) 41 42A.3.6 Process Topologies C Bindings 43 44int MPI\_Cart\_coords(MPI\_Comm comm, int rank, int maxdims, int coords[]) 45int MPI\_Cart\_create(MPI\_Comm comm\_old, int ndims, const int dims[], 46const int periods[], int reorder, MPI\_Comm \*comm\_cart) 4748

ANNEX A. LANGUAGE BINDINGS SUMMARY

900

int	<pre>MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],</pre>	1 2
int	<pre>MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],</pre>	3 4 5
int	<pre>MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)</pre>	6
int	<pre>MPI_Cart_shift(MPI_Comm comm, int direction, int disp,</pre>	7 8 9
int	<pre>MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)</pre>	10
int	MPI_Cartdim_get(MPI_Comm comm, int *ndims)	11 12
	MPI_Dims_create(int nnodes, int ndims, int dims[])	13
		14
int	<pre>MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],</pre>	15 16
	const int weights[], MPI_Info info, int reorder,	17
	MPI_Comm *comm_dist_graph)	18
int	MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,	19
	const int sources[], const int sourceweights[], int outdegree,	20
	<pre>const int destinations[], const int destweights[],</pre>	21
	<pre>MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)</pre>	22 23
int	<pre>MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],</pre>	23 24
	int sourceweights[], int maxoutdegree, int destinations[],	25
	<pre>int destweights[])</pre>	26
int	MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,	27
	int *outdegree, int *weighted)	28
		29
TUC	<pre>MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],</pre>	30 31
		32
int	<pre>MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],</pre>	33
	<pre>int edges[])</pre>	34
int	<pre>MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],</pre>	35
	<pre>const int edges[], int *newrank)</pre>	36
int	MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,	37
	int neighbors[])	38 39
int	MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)	40
		41
int	<pre>MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)</pre>	42
int	<pre>MPI_Ineighbor_allgather(const void *sendbuf, int sendcount,</pre>	43
	MPI_Datatype sendtype, void *recvbuf, int recvcount,	44
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>	45 46
int	<pre>MPI_Ineighbor_allgather_c(const void *sendbuf, MPI_Count sendcount,</pre>	47
	<pre>MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	48

1		<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>
2	int MPT Tne	<pre>ighbor_allgatherv(const void *sendbuf, int sendcount,</pre>
3		MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
4 5		<pre>const int displs[], MPI_Datatype recvtype, MPI_Comm comm,</pre>
6		MPI_Request *request)
7		in the all and have due to the MDT Count and another
8	int MPI_ine	<pre>ighbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,</pre>
9		const MPI_Count recvcounts[], const MPI_Aint displs[],
10 11		MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
12	int MPI_Ine	<pre>ighbor_alltoall(const void *sendbuf, int sendcount,</pre>
13		MPI_Datatype sendtype, void *recvbuf, int recvcount,
14		MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
15	int MPT The	<pre>ighbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,</pre>
16	int in i_inc	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
17		MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
18		
19	int MPI_Ine	<pre>ighbor_alltoallv(const void *sendbuf, const int sendcounts[],</pre>
20		<pre>const int sdispls[], MPI_Datatype sendtype, void *recvbuf,</pre>
21		<pre>const int recvcounts[], const int rdispls[], MDL Detetures recorder [MDL Comm some MDL Detectors]</pre>
22		<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>
23	int MPI_Ine	<pre>ighbor_alltoallv_c(const void *sendbuf,</pre>
24 25		<pre>const MPI_Count sendcounts[], const MPI_Aint sdispls[],</pre>
25 26		MPI_Datatype sendtype, void *recvbuf,
20		<pre>const MPI_Count recvcounts[], const MPI_Aint rdispls[],</pre>
28		<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)</pre>
29	int MPI_Ine	<pre>ighbor_alltoallw(const void *sendbuf, const int sendcounts[],</pre>
30		<pre>const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],</pre>
31		<pre>void *recvbuf, const int recvcounts[],</pre>
32		<pre>const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],</pre>
33		MPI_Comm comm, MPI_Request *request)
34	int MPI Ine	ighbor_alltoallw_c(const void *sendbuf,
35	_	<pre>const MPI_Count sendcounts[], const MPI_Aint sdispls[],</pre>
36		<pre>const MPI_Datatype sendtypes[], void *recvbuf,</pre>
37		<pre>const MPI_Count recvcounts[], const MPI_Aint rdispls[],</pre>
38		<pre>const MPI_Datatype recvtypes[], MPI_Comm comm,</pre>
39		MPI_Request *request)
40 41	int MPT Noi	<pre>ghbor_allgather(const void *sendbuf, int sendcount,</pre>
41 42	THE HET MET	MPI_Datatype sendtype, void *recvbuf, int recvcount,
42		MPI_Datatype recvtype, MPI_Comm comm)
43		
45	int MPI_Nei	<pre>ghbor_allgather_c(const void *sendbuf, MPI_Count sendcount,</pre>
46		MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,
47		MPI_Datatype recvtype, MPI_Comm comm)
48	int MPI_Nei	<pre>ghbor_allgather_init(const void *sendbuf, int sendcount,</pre>

	MPI_Datatype sendtype, void *recvbuf, int recvcount,	1
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	2
	MPI_Request *request)	3
		4
int MPI_Neig	<pre>hbor_allgather_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	5
	MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,	6
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	7
	MPI_Request *request)	8
int MPI Neig	<pre>hbor_allgatherv(const void *sendbuf, int sendcount,</pre>	9
_ 0	<pre>MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],</pre>	10
	const int displs[], MPI_Datatype recvtype, MPI_Comm comm)	11
		12
int MPI_Neig	<pre>hbor_allgatherv_c(const void *sendbuf, MPI_Count sendcount,</pre>	13
	MPI_Datatype sendtype, void *recvbuf,	14
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>	15
	MPI_Datatype recvtype, MPI_Comm comm)	16
int MPT Neig	<pre>hbor_allgatherv_init(const void *sendbuf, int sendcount,</pre>	17
1110 111 1_1016	MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],	18
	const int displs[], MPI_Datatype recvtype, MPI_Comm comm,	19
	MPI_Info info, MPI_Request *request)	20
	In 1_1110 1110, In 1_Request (request)	21
int MPI_Neig	<pre>hbor_allgatherv_init_c(const void *sendbuf,</pre>	22
	<pre>MPI_Count sendcount, MPI_Datatype sendtype, void *recvbuf,</pre>	23
	<pre>const MPI_Count recvcounts[], const MPI_Aint displs[],</pre>	24
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,</pre>	25
	MPI_Request *request)	26
int MPT Neig	<pre>hbor_alltoall(const void *sendbuf, int sendcount,</pre>	27
int mi_weig	MPI_Datatype sendtype, void *recvbuf, int recvcount,	28
	MPI_Datatype recvtype, MPI_Comm comm)	29
		30
int MPI_Neig	<pre>hbor_alltoall_c(const void *sendbuf, MPI_Count sendcount,</pre>	31
	<pre>MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	32
	MPI_Datatype recvtype, MPI_Comm comm)	33
int MPT Neig	<pre>hbor_alltoall_init(const void *sendbuf, int sendcount,</pre>	34
Int In I_Neig	MPI_Datatype sendtype, void *recvbuf, int recvcount,	35
	MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,	36
	MPI_Request *request)	37
	III 1_Request #request)	38
int MPI_Neig	<pre>hbor_alltoall_init_c(const void *sendbuf, MPI_Count sendcount,</pre>	39
	<pre>MPI_Datatype sendtype, void *recvbuf, MPI_Count recvcount,</pre>	40
	<pre>MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,</pre>	41
	MPI_Request *request)	42
int MDT Noig	<pre>hbor_alltoallv(const void *sendbuf, const int sendcounts[],</pre>	43
THE WIT WEIG	const int sdispls[], MPI_Datatype sendtype, void *recvbuf,	44
	const int recvcounts[], const int rdispls[],	45
	MPI_Datatype recvtype, MPI_Comm comm)	46
	In I_Datacype recycype, In I_comm comm)	47
int MPI_Neig	<pre>hbor_alltoallv_c(const void *sendbuf,</pre>	48

```
1
                   const MPI_Count sendcounts[], const MPI_Aint sdispls[],
\mathbf{2}
                   MPI_Datatype sendtype, void *recvbuf,
3
                   const MPI_Count recvcounts[], const MPI_Aint rdispls[],
4
                   MPI_Datatype recvtype, MPI_Comm comm)
5
     int MPI_Neighbor_alltoallv_init(const void *sendbuf,
6
                   const int sendcounts[], const int sdispls[],
7
                   MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
8
                   const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
9
                   MPI_Info info, MPI_Request *request)
10
11
     int MPI_Neighbor_alltoallv_init_c(const void *sendbuf,
12
                   const MPI_Count sendcounts[], const MPI_Aint sdispls[],
13
                   MPI_Datatype sendtype, void *recvbuf,
14
                   const MPI_Count recvcounts[], const MPI_Aint rdispls[],
15
                   MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
16
                   MPI_Request *request)
17
     int MPI_Neighbor_alltoallw(const void *sendbuf, const int sendcounts[],
18
                   const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
19
                   void *recvbuf, const int recvcounts[],
20
                   const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
21
                   MPI_Comm comm)
22
23
     int MPI_Neighbor_alltoallw_c(const void *sendbuf,
24
                   const MPI_Count sendcounts[], const MPI_Aint sdispls[],
25
                   const MPI_Datatype sendtypes[], void *recvbuf,
26
                   const MPI_Count recvcounts[], const MPI_Aint rdispls[],
27
                   const MPI_Datatype recvtypes[], MPI_Comm comm)
28
     int MPI_Neighbor_alltoallw_init(const void *sendbuf,
29
                   const int sendcounts[], const MPI_Aint sdispls[],
30
                   const MPI_Datatype sendtypes[], void *recvbuf,
31
                   const int recvcounts[], const MPI_Aint rdispls[],
32
                   const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
33
                   MPI_Request *request)
34
35
     int MPI_Neighbor_alltoallw_init_c(const void *sendbuf,
36
                   const MPI_Count sendcounts[], const MPI_Aint sdispls[],
37
                   const MPI_Datatype sendtypes[], void *recvbuf,
38
                   const MPI_Count recvcounts[], const MPI_Aint rdispls[],
39
                   const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
40
                   MPI_Request *request)
41
     int MPI_Topo_test(MPI_Comm comm, int *status)
42
43
44
     A.3.7 MPI Environmental Management C Bindings
45
46
     int MPI_Add_error_class(int *errorclass)
47
     int MPI_Add_error_code(int errorclass, int *errorcode)
48
```

<pre>int MPI_Add_error_string(int errorcode, const char *string)</pre>	1
<pre>int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)</pre>	2
<pre>int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)</pre>	4
<pre>int MPI_Comm_create_errhandler(</pre>	5 6 7 8
<pre>int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)</pre>	9
<pre>int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)</pre>	10 11
<pre>int MPI_Errhandler_free(MPI_Errhandler *errhandler)</pre>	12
<pre>int MPI_Error_class(int errorcode, int *errorclass)</pre>	13 14
<pre>int MPI_Error_string(int errorcode, char *string, int *resultlen)</pre>	15
int MPI_File_call_errhandler(MPI_File fh, int errorcode)	16 17
int MPI_File_create_errhandler(	18
MPI_File_errhandler_function *file_errhandler_fn, MPI_Errhandler *errhandler)	19 20
<pre>int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)</pre>	21 22
<pre>int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)</pre>	23
<pre>int MPI_Free_mem(void *base)</pre>	24 25
<pre>int MPI_Get_library_version(char *version, int *resultlen)</pre>	26
<pre>int MPI_Get_processor_name(char *name, int *resultlen)</pre>	27 28
<pre>int MPI_Get_version(int *version, int *subversion)</pre>	29
	30
<pre>int MPI_Session_call_errhandler(MPI_Session session, int errorcode)</pre>	31 32
int MPI_Session_create_errhandler(	33
<pre>MPI_Session_errhandler_function *session_errhandler_fn, MPI_Errhandler *errhandler)</pre>	34
	35 36
<pre>int MPI_Session_get_errhandler(MPI_Session session,</pre>	37
	38
<pre>int MPI_Session_set_errhandler(MPI_Session session,</pre>	39
	40 41
<pre>int MPI_Win_call_errhandler(MPI_Win win, int errorcode)</pre>	42
int MPI_Win_create_errhandler(	43
<pre>MPI_Win_errhandler_function *win_errhandler_fn, MPI_Errhandler *errhandler)</pre>	44 45
	45 46
<pre>int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)</pre>	47
<pre>int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)</pre>	48

```
1
     double MPI_Wtick(void)
\mathbf{2}
     double MPI_Wtime(void)
3
4
\mathbf{5}
     A.3.8 The Info Object C Bindings
6
     int MPI_Info_create(MPI_Info *info)
7
8
     int MPI_Info_create_env(int argc, char argv[], MPI_Info *info)
9
     int MPI_Info_delete(MPI_Info info, const char *key)
10
11
     int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
12
     int MPI_Info_free(MPI_Info *info)
13
14
     int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
15
16
     int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
17
     int MPI_Info_get_string(MPI_Info info, const char *key, int *buflen,
18
                   char *value, int *flag)
19
20
     int MPI_Info_set(MPI_Info info, const char *key, const char *value)
21
22
     A.3.9 Process Creation and Management C Bindings
23
^{24}
     int MPI_Abort(MPI_Comm comm, int errorcode)
25
     int MPI_Close_port(const char *port_name)
26
27
     int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
28
                   MPI_Comm comm, MPI_Comm *newcomm)
29
     int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
30
                   MPI_Comm comm, MPI_Comm *newcomm)
^{31}
32
     int MPI_Comm_disconnect(MPI_Comm *comm)
33
34
     int MPI_Comm_get_parent(MPI_Comm *parent)
35
     int MPI_Comm_join(int fd, MPI_Comm *intercomm)
36
37
     int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
                   MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm,
38
                   int array_of_errcodes[])
39
40
     int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
41
                   char **array_of_argv[], const int array_of_maxprocs[],
42
                   const MPI_Info array_of_info[], int root, MPI_Comm comm,
43
                   MPI_Comm *intercomm, int array_of_errcodes[])
44
45
     int MPI_Finalize(void)
46
     int MPI_Finalized(int *flag)
47
48
     int MPI_Init(int *argc, char ***argv)
```

int	<pre>MPI_Init_thread(int *argc, char ***argv, int required, int *provided)</pre>	1
int	MPI_Initialized(int *flag)	2 3
int	MPI_Is_thread_main(int *flag)	4
int	<pre>MPI_Lookup_name(const char *service_name, MPI_Info info,</pre>	5 6 7
int	MPI_Open_port(MPI_Info info, char *port_name)	8
		9
int	<pre>MPI_Publish_name(const char *service_name, MPI_Info info,</pre>	10 11
int	MPI_Query_thread(int *provided)	12 13
int	MPI_Session_finalize(MPI_Session *session)	13
int	<pre>MPI_Session_get_info(MPI_Session session, MPI_Info *info_used)</pre>	15 16
int	<pre>MPI_Session_get_nth_pset(MPI_Session session, MPI_Info info, int n,</pre>	17 18
int	<pre>MPI_Session_get_num_psets(MPI_Session session, MPI_Info info,</pre>	19 20 21
int	<pre>MPI_Session_get_pset_info(MPI_Session session, const char *pset_name,</pre>	21 22 23
int	<pre>MPI_Session_init(MPI_Info info, MPI_Errhandler errhandler,</pre>	24 25 26
int	<pre>MPI_Unpublish_name(const char *service_name, MPI_Info info,</pre>	20 27 28 29
A.3.	10 One-Sided Communications C Bindings	30 31
int	<pre>MPI_Accumulate(const void *origin_addr, int origin_count,</pre>	32 33 34 35
int	<pre>MPI_Accumulate_c(const void *origin_addr, MPI_Count origin_count,</pre>	36 37 38 39 40
int	<pre>MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr, void *result_addr, MPI_Datatype datatype, int target_rank, MPI_Aint target_disp, MPI_Win win)</pre>	41 42 43 44
int	<pre>MPI_Fetch_and_op(const void *origin_addr, void *result_addr,</pre>	44 45 46 47 48

```
1
     int MPI_Get(void *origin_addr, int origin_count,
\mathbf{2}
                   MPI_Datatype origin_datatype, int target_rank,
3
                   MPI_Aint target_disp, int target_count,
4
                   MPI_Datatype target_datatype, MPI_Win win)
5
     int MPI_Get_accumulate(const void *origin_addr, int origin_count,
6
                   MPI_Datatype origin_datatype, void *result_addr,
7
                   int result_count, MPI_Datatype result_datatype,
8
                   int target_rank, MPI_Aint target_disp, int target_count,
9
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
10
11
     int MPI_Get_accumulate_c(const void *origin_addr, MPI_Count origin_count,
12
                  MPI_Datatype origin_datatype, void *result_addr,
13
                  MPI_Count result_count, MPI_Datatype result_datatype,
14
                   int target_rank, MPI_Aint target_disp, MPI_Count target_count,
15
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
16
     int MPI_Get_c(void *origin_addr, MPI_Count origin_count,
17
                   MPI_Datatype origin_datatype, int target_rank,
18
                   MPI_Aint target_disp, MPI_Count target_count,
19
                   MPI_Datatype target_datatype, MPI_Win win)
20
21
     int MPI_Put(const void *origin_addr, int origin_count,
22
                   MPI_Datatype origin_datatype, int target_rank,
23
                   MPI_Aint target_disp, int target_count,
^{24}
                   MPI_Datatype target_datatype, MPI_Win win)
25
     int MPI_Put_c(const void *origin_addr, MPI_Count origin_count,
26
                  MPI_Datatype origin_datatype, int target_rank,
27
                   MPI_Aint target_disp, MPI_Count target_count,
28
                   MPI_Datatype target_datatype, MPI_Win win)
29
30
     int MPI_Raccumulate(const void *origin_addr, int origin_count,
^{31}
                   MPI_Datatype origin_datatype, int target_rank,
32
                   MPI_Aint target_disp, int target_count,
33
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
34
                   MPI_Request *request)
35
     int MPI_Raccumulate_c(const void *origin_addr, MPI_Count origin_count,
36
                   MPI_Datatype origin_datatype, int target_rank,
37
                   MPI_Aint target_disp, MPI_Count target_count,
38
                   MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
39
                   MPI_Request *request)
40
41
     int MPI_Rget(void *origin_addr, int origin_count,
42
                  MPI_Datatype origin_datatype, int target_rank,
43
                   MPI_Aint target_disp, int target_count,
44
                   MPI_Datatype target_datatype, MPI_Win win,
45
                  MPI_Request *request)
46
     int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
47
                  MPI_Datatype origin_datatype, void *result_addr,
48
```

	<pre>int result_count, MPI_Datatype result_datatype,</pre>	1
	int target_rank, MPI_Aint target_disp, int target_count,	2
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,	3
	MPI_Request *request)	4
		5
int MPI_Rget	<pre>c_accumulate_c(const void *origin_addr, MPI_Count origin_count,</pre>	6
	MPI_Datatype origin_datatype, void *result_addr,	7
	MPI_Count result_count, MPI_Datatype result_datatype,	8
	<pre>int target_rank, MPI_Aint target_disp, MPI_Count target_count,</pre>	9
	MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,	10
	MPI_Request *request)	11
int MPI_Rget	<pre>c_c(void *origin_addr, MPI_Count origin_count,</pre>	12
- 0	MPI_Datatype origin_datatype, int target_rank,	13
	MPI_Aint target_disp, MPI_Count target_count,	14
	MPI_Datatype target_datatype, MPI_Win win,	15
	MPI_Request *request)	16
		17
int MPI_Rput	c(const void *origin_addr, int origin_count,	18
	MPI_Datatype origin_datatype, int target_rank,	19
	<pre>MPI_Aint target_disp, int target_count,</pre>	20
	MPI_Datatype target_datatype, MPI_Win win,	21
	MPI_Request *request)	22
int MPT Rout	<pre>c_c(const void *origin_addr, MPI_Count origin_count,</pre>	23
	MPI_Datatype origin_datatype, int target_rank,	24
	MPI_Aint target_disp, MPI_Count target_count,	25
	MPI_Datatype target_datatype, MPI_Win win,	26
	MPI_Request *request)	27
		28
int MPI_Win_	_allocate(MPI_Aint size, int disp_unit, MPI_Info info,	29
	<pre>MPI_Comm comm, void *baseptr, MPI_Win *win)</pre>	30
int MPT Win	_allocate_c(MPI_Aint size, MPI_Aint disp_unit, MPI_Info info,	31
1110 111 1_w111_	MPI_Comm comm, void *baseptr, MPI_Win *win)	32
		33
int MPI_Win_	_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,	34
	<pre>MPI_Comm comm, void *baseptr, MPI_Win *win)</pre>	35
int MDT Win	_allocate_shared_c(MPI_Aint size, MPI_Aint disp_unit,	36
IIIC PILL_WIII_	MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)	37
	MI_INTO INTO, MI_COMM COMM, VOId *Dasepti, MI_WIN *WIN/	38
int MPI_Win_	_attach(MPI_Win win, void *base, MPI_Aint size)	39
int MDT Win	_complete(MPI_Win win)	40
IIIC III I_WIII_	combrece(uni_win win)	41
int MPI_Win_	<pre>create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,</pre>	42
	MPI_Comm comm, MPI_Win *win)	43
int MPT Win	<pre>_create_c(void *base, MPI_Aint size, MPI_Aint disp_unit,</pre>	44
THE BUT WIT	MPI_Info info, MPI_Comm comm, MPI_Win *win)	45
	THE THE THE , THE COMM COMM, THE WILL TWILL	46
int MPI_Win_	_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)	47
		48

1	<pre>int MPI_Win_detach(MPI_Win win, const void *base)</pre>
2 3	<pre>int MPI_Win_fence(int assert, MPI_Win win)</pre>
4	<pre>int MPI_Win_flush(int rank, MPI_Win win)</pre>
5 6	<pre>int MPI_Win_flush_all(MPI_Win win)</pre>
7	int MPI_Win_flush_local(int rank, MPI_Win win)
8 9	<pre>int MPI_Win_flush_local_all(MPI_Win win)</pre>
10	<pre>int MPI_Win_free(MPI_Win *win)</pre>
11 12	<pre>int MPI_Win_get_group(MPI_Win win, MPI_Group *group)</pre>
13	int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
14 15	<pre>int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)</pre>
16	<pre>int MPI_Win_lock_all(int assert, MPI_Win win)</pre>
17 18	<pre>int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)</pre>
19	int MPI_Win_set_info(MPI_Win win, MPI_Info info)
20 21	<pre>int MPI_Win_set_Info(MPI_Win win, MPI_Info Info) int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,</pre>
22	int *disp_unit, void *baseptr)
23 24	int MPI_Win_shared_query_c(MPI_Win win, int rank, MPI_Aint *size,
24 25	<pre>MPI_Aint *disp_unit, void *baseptr)</pre>
26	<pre>int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)</pre>
27 28	<pre>int MPI_Win_sync(MPI_Win win)</pre>
29	<pre>int MPI_Win_test(MPI_Win win, int *flag)</pre>
30 31	<pre>int MPI_Win_unlock(int rank, MPI_Win win)</pre>
32	<pre>int MPI_Win_unlock_all(MPI_Win win)</pre>
33 34	int MPI_Win_wait(MPI_Win win)
35	
$\frac{36}{37}$	A.3.11 External Interfaces C Bindings
38	<pre>int MPI_Grequest_complete(MPI_Request request)</pre>
39 40	<pre>int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,</pre>
41	<pre>MPI_Grequest_free_function *free_fn, MPI_Grequest_cancel_function *cancel_fn, void *extra_state,</pre>
42 43	MPI_Request *request)
44	<pre>int MPI_Status_set_cancelled(MPI_Status *status, int flag)</pre>
45 46	<pre>int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,</pre>
40	int count)
48	

<pre>int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,</pre>	1 2 3
A.3.12 I/O C Bindings	4 5
<pre>int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count,</pre>	6 7 8
<pre>int MPI_CONVERSION_FN_NULL_C(void *userbuf, MPI_Datatype datatype,</pre>	9 10 11
<pre>int MPI_File_close(MPI_File *fh)</pre>	12 13
<pre>int MPI_File_delete(const char *filename, MPI_Info info)</pre>	14
<pre>int MPI_File_get_amode(MPI_File fh, int *amode)</pre>	15 16
<pre>int MPI_File_get_atomicity(MPI_File fh, int *flag)</pre>	17
<pre>int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,</pre>	18 19 20
<pre>int MPI_File_get_group(MPI_File fh, MPI_Group *group)</pre>	21
<pre>int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)</pre>	22 23
<pre>int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)</pre>	24
<pre>int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)</pre>	25 26
<pre>int MPI_File_get_size(MPI_File fh, MPI_Offset *size)</pre>	27
<pre>int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,</pre>	28 29 30
<pre>int MPI_File_get_type_extent_c(MPI_File fh, MPI_Datatype datatype,</pre>	31 32
<pre>int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep)</pre>	33 34 35
<pre>int MPI_File_iread(MPI_File fh, void *buf, int count,</pre>	36 37
<pre>int MPI_File_iread_all(MPI_File fh, void *buf, int count,</pre>	38 39 40
<pre>int MPI_File_iread_all_c(MPI_File fh, void *buf, MPI_Count count,</pre>	41 42 43
<pre>int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,</pre>	43 44 45
<pre>int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	46 47 48

1 2	int	<pre>MPI_File_iread_at_all_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>
3 4 5	int	<pre>MPI_File_iread_at_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>
6 7	int	<pre>MPI_File_iread_c(MPI_File fh, void *buf, MPI_Count count,</pre>
8 9 10	int	<pre>MPI_File_iread_shared(MPI_File fh, void *buf, int count,</pre>
11 12	int	<pre>MPI_File_iread_shared_c(MPI_File fh, void *buf, MPI_Count count,</pre>
13 14 15	int	<pre>MPI_File_iwrite(MPI_File fh, const void *buf, int count,</pre>
16 17	int	<pre>MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,</pre>
18 19 20	int	<pre>MPI_File_iwrite_all_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Request *request)</pre>
21 22	int	<pre>MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
23 24 25	int	<pre>MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
26 27 28	int	<pre>MPI_File_iwrite_at_all_c(MPI_File fh, MPI_Offset offset,</pre>
29 30 31	int	<pre>MPI_File_iwrite_at_c(MPI_File fh, MPI_Offset offset, const void *buf,</pre>
32 33 34	int	<pre>MPI_File_iwrite_c(MPI_File fh, const void *buf, MPI_Count count,</pre>
35 36	int	<pre>MPI_File_iwrite_shared(MPI_File fh, const void *buf, int count,</pre>
37 38 39	int	<pre>MPI_File_iwrite_shared_c(MPI_File fh, const void *buf, MPI_Count count,</pre>
40 41	int	<pre>MPI_File_open(MPI_Comm comm, const char *filename, int amode,</pre>
42 43	int	<pre>MPI_File_preallocate(MPI_File fh, MPI_Offset size)</pre>
44 45	int	<pre>MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,</pre>
46 47 48	int	<pre>MPI_File_read_all(MPI_File fh, void *buf, int count,</pre>

int	<pre>MPI_File_read_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)</pre>	1 2
int	<pre>MPI_File_read_all_begin_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype)</pre>	3 4 5
int	<pre>MPI_File_read_all_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	6 7
int	<pre>MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	8 9
int	<pre>MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)</pre>	10 11
int	<pre>MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,</pre>	12 13 14
int	<pre>MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,</pre>	15 16
int	<pre>MPI_File_read_at_all_begin_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>	17 18 19
int	<pre>MPI_File_read_at_all_c(MPI_File fh, MPI_Offset offset, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	20 21 22
int	<pre>MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	23
int	<pre>MPI_File_read_at_c(MPI_File fh, MPI_Offset offset, void *buf,</pre>	24 25 26
int	<pre>MPI_File_read_c(MPI_File fh, void *buf, MPI_Count count,</pre>	27 28
int	<pre>MPI_File_read_ordered(MPI_File fh, void *buf, int count,</pre>	29 30 31
int	<pre>MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,</pre>	32 33
int	<pre>MPI_File_read_ordered_begin_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype)</pre>	34 35 36
int	<pre>MPI_File_read_ordered_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	37 38
int	<pre>MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)</pre>	39 40
int	MPI_File_read_shared(MPI_File fh, void *buf, int count,	41
	MPI_Datatype datatype, MPI_Status *status)	42 43
int	<pre>MPI_File_read_shared_c(MPI_File fh, void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	43 44 45
int	MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)	46
int	MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)	47 48

```
1
     int MPI_File_set_atomicity(MPI_File fh, int flag)
\mathbf{2}
     int MPI_File_set_info(MPI_File fh, MPI_Info info)
3
4
     int MPI_File_set_size(MPI_File fh, MPI_Offset size)
5
     int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
6
                   MPI_Datatype filetype, const char *datarep, MPI_Info info)
7
8
     int MPI_File_sync(MPI_File fh)
9
     int MPI_File_write(MPI_File fh, const void *buf, int count,
10
                   MPI_Datatype datatype, MPI_Status *status)
11
12
     int MPI_File_write_all(MPI_File fh, const void *buf, int count,
13
                   MPI_Datatype datatype, MPI_Status *status)
14
     int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
15
                   MPI_Datatype datatype)
16
17
     int MPI_File_write_all_begin_c(MPI_File fh, const void *buf,
18
                   MPI_Count count, MPI_Datatype datatype)
19
     int MPI_File_write_all_c(MPI_File fh, const void *buf, MPI_Count count,
20
                   MPI_Datatype datatype, MPI_Status *status)
21
22
     int MPI_File_write_all_end(MPI_File fh, const void *buf,
23
                   MPI_Status *status)
^{24}
     int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
25
26
                   int count, MPI_Datatype datatype, MPI_Status *status)
27
     int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
28
                   int count, MPI_Datatype datatype, MPI_Status *status)
29
     int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
30
^{31}
                   const void *buf, int count, MPI_Datatype datatype)
32
     int MPI_File_write_at_all_begin_c(MPI_File fh, MPI_Offset offset,
33
                   const void *buf, MPI_Count count, MPI_Datatype datatype)
34
     int MPI_File_write_at_all_c(MPI_File fh, MPI_Offset offset,
35
36
                   const void *buf, MPI_Count count, MPI_Datatype datatype,
37
                   MPI_Status *status)
38
     int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
39
                   MPI_Status *status)
40
41
     int MPI_File_write_at_c(MPI_File fh, MPI_Offset offset, const void *buf,
42
                   MPI_Count count, MPI_Datatype datatype, MPI_Status *status)
43
     int MPI_File_write_c(MPI_File fh, const void *buf, MPI_Count count,
44
                  MPI_Datatype datatype, MPI_Status *status)
45
46
     int MPI_File_write_ordered(MPI_File fh, const void *buf, int count,
47
                   MPI_Datatype datatype, MPI_Status *status)
48
```

int	<pre>MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,</pre>	1 2
int	MPI_File_write_ordered_begin_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype)	3 4
int	<pre>MPI_File_write_ordered_c(MPI_File fh, const void *buf, MPI_Count count, MPI_Datatype datatype, MPI_Status *status)</pre>	5 6 7
int	MPI_File_write_ordered_end(MPI_File fh, const void *buf, MPI_Status *status)	8 9 10
int	MPI_File_write_shared(MPI_File fh, const void *buf, int count, MPI_Datatype datatype, MPI_Status *status)	11 12
int	<pre>MPI_File_write_shared_c(MPI_File fh, const void *buf, MPI_Count count,</pre>	13 14 15
int	<pre>MPI_Register_datarep(const char *datarep, MPI_Datarep_conversion_function *read_conversion_fn, MPI_Datarep_conversion_function *write_conversion_fn, MPI_Datarep_extent_function *dtype_file_extent_fn, void *extra_state)</pre>	16 17 18 19 20
int	<pre>MPI_Register_datarep_c(const char *datarep,</pre>	21 22 23 24 25 26
A.3.	.3 Language Bindings C Bindings	27 28 29
MPI_	Fint MPI_Comm_c2f(MPI_Comm comm)	29 30
	Comm MPI_Comm_f2c(MPI_Fint comm)	31
	Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)	32 33
		34
	Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)	35
MPI_	Fint MPI_File_c2f(MPI_File file)	36 37
MPI_	File MPI_File_f2c(MPI_Fint file)	38
MPI_	Fint MPI_Group_c2f(MPI_Group group)	39
MPI	Group MPI_Group_f2c(MPI_Fint group)	40 41
	Fint MPI_Info_c2f(MPI_Info info)	41
		43
MPI_	Info MPI_Info_f2c(MPI_Fint info)	44
MPI_	Fint MPI_Message_c2f(MPI_Message message)	45 46
MPI_	Message MPI_Message_f2c(MPI_Fint message)	47
		48

1	MPI_Fint MPI_Op_c2f(MPI_Op op)
$\frac{2}{3}$	MPI_Op_MPI_Op_f2c(MPI_Fint op)
4	MPI_Fint MPI_Request_c2f(MPI_Request request)
5 6	MPI_Request MPI_Request_f2c(MPI_Fint request)
7	MPI_Fint MPI_Session_c2f(MPI_Session session)
8 9	MPI_Session MPI_Session_f2c(MPI_Fint session)
10	<pre>int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)</pre>
11 12 13	<pre>int MPI_Status_c2f08(const MPI_Status *c_status,</pre>
14 15 16	<pre>int MPI_Status_f082c(const MPI_F08_status *f08_status,</pre>
17	<pre>int MPI_Status_f082f(const MPI_F08_status *f08_status, MPI_Fint *f_status)</pre>
18 19	<pre>int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)</pre>
20	<pre>int MPI_Status_f2f08(const MPI_Fint *f_status, MPI_F08_status *f08_status)</pre>
21 22	MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
23	<pre>int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)</pre>
$\frac{24}{25}$	<pre>int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)</pre>
26	<pre>int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)</pre>
27 28	<pre>MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)</pre>
29	<pre>int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)</pre>
30 31	MPI_Fint MPI_Win_c2f(MPI_Win win)
32 33	MPI_Win MPI_Win_f2c(MPI_Fint win)
34	
35 36	A.3.14 Tools / Profiling Interface C Bindings
37	<pre>int MPI_Pcontrol(const int level,)</pre>
38 39	A.3.15 Tools / MPI Tool Information Interface C Bindings
40	<pre>int MPI_T_category_changed(int *update_number)</pre>
41 42	<pre>int MPI_T_category_get_categories(int cat_index, int len, int indices[])</pre>
43	<pre>int MPI_T_category_get_cvars(int cat_index, int len, int indices[])</pre>
44 45	<pre>int MPI_T_category_get_events(int cat_index, int len, int indices[])</pre>
46	<pre>int MPI_T_category_get_index(const char *name, int *cat_index)</pre>
47 48	

int	<pre>MPI_T_category_get_info(int cat_index, char *name, int *name_len,</pre>	1 $2$
	<pre>int *num_categories)</pre>	3
int	MPI_T_category_get_num(int *num_cat)	4
		5 6
int	<pre>MPI_T_category_get_num_events(int cat_index, int *num_events)</pre>	7
int	<pre>MPI_T_category_get_pvars(int cat_index, int len, int indices[])</pre>	8
int	<pre>MPI_T_cvar_get_index(const char *name, int *cvar_index)</pre>	9
int	<pre>MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,</pre>	10 11
	int *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype,	12
	char *desc, int *desc_len, int *bind, int *scope)	13
int	<pre>MPI_T_cvar_get_num(int *num_cvar)</pre>	14
		15
int	<pre>MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,</pre>	16
	<pre>MPI_T_cvar_handle *handle, int *count)</pre>	17
int	<pre>MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)</pre>	18
int	<pre>MPI_T_cvar_read(MPI_T_cvar_handle handle, void *buf)</pre>	19 20
		21
int	<pre>MPI_T_cvar_write(MPI_T_cvar_handle handle, const void *buf)</pre>	22
int	<pre>MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,</pre>	23
	int *name_len)	24
int	MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value,	25
	char *name, int *name_len)	26
int	MPI_T_event_callback_get_info(	27 28
THE	MPI_T_event_registration event_registration,	29
	MPI_T_cb_safety cb_safety, MPI_Info *info_used)	30
		31
int	MPI_T_event_callback_set_info(	32
	<pre>MPI_T_event_registration event_registration, MPI_T_cb_safety cb_safety, MPI_Info info)</pre>	33
		34
int	<pre>MPI_T_event_copy(MPI_T_event_instance event_instance, void *buffer)</pre>	35
int	<pre>MPI_T_event_get_index(const char *name, int *event_index)</pre>	36 37
		38
int	<pre>MPI_T_event_get_info(int event_index, char *name, int *name_len,</pre>	39
	MPI_Aint array_of_displacements[], int *num_elements,	40
	MPI_T_enum *enumtype, MPI_Info *info, char *desc,	41
	int *desc_len, int *bind)	42
		43
int	MPI_T_event_get_num(int *num_events)	44
int	<pre>MPI_T_event_get_source(MPI_T_event_instance event_instance,</pre>	45
	<pre>int *source_index)</pre>	46
		47 48
		40

```
1
     int MPI_T_event_get_timestamp(MPI_T_event_instance event_instance,
\mathbf{2}
                   MPI_Count *event_timestamp)
3
     int MPI_T_event_handle_alloc(int event_index, void *obj_handle,
4
                   MPI_Info info, MPI_T_event_registration *event_registration)
5
6
     int MPI_T_event_handle_free(MPI_T_event_registration event_registration,
7
                   void *user_data,
8
                   MPI_T_event_free_cb_function free_cb_function)
9
     int MPI_T_event_handle_get_info(
10
                   MPI_T_event_registration event_registration,
11
                   MPI_Info *info_used)
12
13
     int MPI_T_event_handle_set_info(
14
                   MPI_T_event_registration event_registration, MPI_Info info)
15
     int MPI_T_event_read(MPI_T_event_instance event_instance,
16
                   int element_index, void *buffer)
17
18
     int MPI_T_event_register_callback(
19
                   MPI_T_event_registration event_registration,
20
                   MPI_T_cb_safety cb_safety, MPI_Info info, void *user_data,
21
                   MPI_T_event_cb_function event_cb_function)
22
     int MPI_T_event_set_dropped_handler(
23
                   MPI_T_event_registration event_registration,
24
                   MPI_T_event_dropped_cb_function dropped_cb_function)
25
26
     int MPI_T_finalize(void)
27
     int MPI_T_init_thread(int required, int *provided)
28
29
     int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)
30
     int MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,
^{31}
                   int *verbosity, int *var_class, MPI_Datatype *datatype,
32
                   MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind,
33
34
                   int *readonly, int *continuous, int *atomic)
35
     int MPI_T_pvar_get_num(int *num_pvar)
36
37
     int MPI_T_pvar_handle_alloc(MPI_T_pvar_session pe_session, int pvar_index,
                   void *obj_handle, MPI_T_pvar_handle *handle, int *count)
38
39
     int MPI_T_pvar_handle_free(MPI_T_pvar_session pe_session,
40
                   MPI_T_pvar_handle *handle)
41
42
     int MPI_T_pvar_read(MPI_T_pvar_session pe_session,
                   MPI_T_pvar_handle handle, void *buf)
43
44
     int MPI_T_pvar_readreset(MPI_T_pvar_session pe_session,
45
                   MPI_T_pvar_handle handle, void *buf)
46
47
     int MPI_T_pvar_reset(MPI_T_pvar_session pe_session,
48
```

MPI_T_pvar_handle handle)	1
<pre>int MPI_T_pvar_session_create(MPI_T_pvar_session *pe_session)</pre>	2 3
<pre>int MPI_T_pvar_session_free(MPI_T_pvar_session *pe_session)</pre>	4
<pre>int MPI_T_pvar_start(MPI_T_pvar_session pe_session,</pre>	5 6 7
<pre>int MPI_T_pvar_stop(MPI_T_pvar_session pe_session,</pre>	8 9
<pre>int MPI_T_pvar_write(MPI_T_pvar_session pe_session,</pre>	10 11 12
<pre>int MPI_T_source_get_info(int source_index, char *name, int *name_len,</pre>	13 14 15 16 17
<pre>int MPI_T_source_get_num(int *num_sources)</pre>	18
<pre>int MPI_T_source_get_timestamp(int source_index, MPI_Count *timestamp)</pre>	19 20
A.3.16 Deprecated C Bindings	21 22
int MPI_Attr_delete(MPI_Comm comm, int keyval)	23
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)	24 25
int MPI_Attr_put(MPI_Comm comm, int keyval, void *attribute_val)	26 27
<pre>int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>	28 29
<pre>int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,</pre>	30 31 32
<pre>int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,</pre>	33 34
<pre>int MPI_Keyval_create(MPI_Copy_function *copy_fn,</pre>	35 36 37 38
<pre>int MPI_Keyval_free(int *keyval)</pre>	39 40
<pre>int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void *extra_state,</pre>	40 41 42
<pre>int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,</pre>	43 44 45 46 47 48

```
A.4 Fortran 2008 Bindings with the mpi_f08 Module
1
\mathbf{2}
     A.4.1 Point-to-Point Communication Fortran 2008 Bindings
3
4
     MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
6
         INTEGER, INTENT(IN) :: count, dest, tag
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Bsend(buf, count, datatype, dest, tag, comm, ierror) !(_c)
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
12
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         INTEGER, INTENT(IN) :: dest, tag
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
19
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
20
         INTEGER, INTENT(IN) :: count, dest, tag
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         TYPE(MPI_Request), INTENT(OUT) :: request
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
27
                   !(_c)
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
28
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
29
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
30
31
         INTEGER, INTENT(IN) :: dest, tag
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Request), INTENT(OUT) :: request
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Buffer_attach(buffer, size, ierror)
36
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
37
         INTEGER, INTENT(IN) :: size
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Buffer_attach(buffer, size, ierror) !(_c)
41
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: size
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Buffer_detach(buffer_addr, size, ierror)
45
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
46
         TYPE(C_PTR), INTENT(OUT) :: buffer_addr
47
         INTEGER, INTENT(OUT) :: size
48
```

920

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Buffer_detach(buffer_addr, size, ierror) !(_c)	2 3
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR TYPE(C_PTR), INTENT(OUT) :: buffer_addr	4
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size	5
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	6
	7
MPI_Cancel(request, ierror)	8
TYPE(MPI_Request), INTENT(IN) :: request	9 10
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
<pre>MPI_Get_count(status, datatype, count, ierror)</pre>	12
TYPE(MPI_Status), INTENT(IN) :: status	13
TYPE(MPI_Datatype), INTENT(IN) :: datatype	14
INTEGER, INTENT(OUT) :: count	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
<pre>MPI_Get_count(status, datatype, count, ierror) !(_c)</pre>	17
TYPE(MPI_Status), INTENT(IN) :: status	18
TYPE(MPI_Datatype), INTENT(IN) :: datatype	19
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	21
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)	22
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	23 24
INTEGER, INTENT(IN) :: count, dest, tag	24 25
TYPE(MPI_Datatype), INTENT(IN) :: datatype	25 26
TYPE(MPI_Comm), INTENT(IN) :: comm	20
TYPE(MPI_Request), INTENT(OUT) :: request	28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	29
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)	30
TYPE(*), DIMENSION(), INTENT(IN), ASYNCHRONOUS :: buf	31
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count	32
TYPE(MPI_Datatype), INTENT(IN) :: datatype	33
INTEGER, INTENT(IN) :: dest, tag	34
TYPE(MPI_Comm), INTENT(IN) :: comm	35
TYPE(MPI_Request), INTENT(OUT) :: request	36
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37
MDT Improhe (acurace to a commention measure atotuce icomer)	38
<pre>MPI_Improbe(source, tag, comm, flag, message, status, ierror) INTEGER, INTENT(IN) :: source, tag</pre>	39
TYPE(MPI_Comm), INTENT(IN) :: comm	40
LOGICAL, INTENT(OUT) :: flag	41 42
TYPE(MPI_Message), INTENT(OUT) :: message	42
TYPE(MPI_Status) :: status	43 44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
	46
MPI_Imrecv(buf, count, datatype, message, request, ierror)	47
TYPE(*), DIMENSION(), ASYNCHRONOUS :: buf	48

```
1
         INTEGER, INTENT(IN) :: count
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Message), INTENT(INOUT) :: message
4
         TYPE(MPI_Request), INTENT(OUT) :: request
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Imrecv(buf, count, datatype, message, request, ierror) !(_c)
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Message), INTENT(INOUT) :: message
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Iprobe(source, tag, comm, flag, status, ierror)
15
         INTEGER, INTENT(IN) :: source, tag
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         LOGICAL, INTENT(OUT) :: flag
18
         TYPE(MPI_Status) :: status
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
22
         INTEGER, INTENT(IN) :: count, source, tag
23
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror) !(_c)
29
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
^{31}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         INTEGER, INTENT(IN) :: source, tag
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         TYPE(MPI_Request), INTENT(OUT) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
37
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
38
         INTEGER, INTENT(IN) :: count, dest, tag
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
^{44}
     MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
45
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    INTEGER, INTENT(IN) :: dest, tag
                                                                                  2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  3
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
                                                                                  5
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                  6
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  7
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                  8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  9
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
                                                                                  13
                                                                                  14
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  15
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  16
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  17
    INTEGER, INTENT(IN) :: dest, tag
                                                                                  18
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  20
                                                                                  21
MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
                                                                                  22
              recvcount, recvtype, source, recvtag, comm, request, ierror)
                                                                                  23
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  24
    INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
                                                                                  25
              recvtag
                                                                                  26
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  27
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  29
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  31
                                                                                  32
MPI_Isendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
                                                                                  33
              recvcount, recvtype, source, recvtag, comm, request, ierror)
                                                                                  34
              !(_c)
                                                                                  35
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  36
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  38
    INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
                                                                                  39
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  41
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                                                                                  44
             comm, request, ierror)
                                                                                  45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  46
    INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
                                                                                  47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Isendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
5
                   comm, request, ierror) !(_c)
6
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
7
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
15
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
16
         INTEGER, INTENT(IN) :: count, dest, tag
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Comm), INTENT(IN) :: comm
19
         TYPE(MPI_Request), INTENT(OUT) :: request
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
22
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
23
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
24
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
25
         INTEGER, INTENT(IN) :: dest, tag
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Mprobe(source, tag, comm, message, status, ierror)
31
         INTEGER, INTENT(IN) :: source, tag
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Message), INTENT(OUT) :: message
34
         TYPE(MPI_Status) :: status
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Mrecv(buf, count, datatype, message, status, ierror)
37
         TYPE(*), DIMENSION(..) :: buf
38
         INTEGER, INTENT(IN) :: count
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Message), INTENT(INOUT) :: message
41
         TYPE(MPI_Status) :: status
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Mrecv(buf, count, datatype, message, status, ierror) !(_c)
45
         TYPE(*), DIMENSION(..) :: buf
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    TYPE(MPI_Message), INTENT(INOUT) :: message
                                                                                   2
    TYPE(MPI_Status) :: status
                                                                                   3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
MPI_Probe(source, tag, comm, status, ierror)
                                                                                   5
    INTEGER, INTENT(IN) :: source, tag
                                                                                   6
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   7
    TYPE(MPI_Status) :: status
                                                                                   8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   9
                                                                                  10
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
                                                                                  11
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
                                                                                  12
                                                                                  13
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  14
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
                                                                                  15
                                                                                  16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  17
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror) !(_c)
                                                                                  18
    TYPE(*), DIMENSION(..) :: buf
                                                                                  19
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  20
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  21
    INTEGER, INTENT(IN) :: source, tag
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  23
    TYPE(MPI_Status) :: status
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
                                                                                  26
MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
                                                                                  27
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  28
    INTEGER, INTENT(IN) :: count, source, tag
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  31
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
                                                                                  34
              !(_c)
                                                                                  35
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  36
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  38
    INTEGER, INTENT(IN) :: source, tag
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   40
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  42
                                                                                  43
MPI_Request_free(request, ierror)
                                                                                  44
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
MPI_Request_get_status(request, flag, status, ierror)
                                                                                  47
    TYPE(MPI_Request), INTENT(IN) :: request
                                                                                  48
```

```
1
         LOGICAL, INTENT(OUT) :: flag
2
         TYPE(MPI_Status) :: status
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
6
         INTEGER, INTENT(IN) :: count, dest, tag
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror) !(_c)
12
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
13
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
14
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
15
         INTEGER, INTENT(IN) :: dest, tag
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
19
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
20
         INTEGER, INTENT(IN) :: count, dest, tag
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         TYPE(MPI_Request), INTENT(OUT) :: request
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
27
                   !(_c)
28
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
29
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
30
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
^{31}
         INTEGER, INTENT(IN) :: dest, tag
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Request), INTENT(OUT) :: request
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror)
36
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
37
         INTEGER, INTENT(IN) :: count, dest, tag
38
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_Send(buf, count, datatype, dest, tag, comm, ierror) !(_c)
43
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
44
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         INTEGER, INTENT(IN) :: dest, tag
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  1
                                                                                  2
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
                                                                                  3
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  4
    INTEGER, INTENT(IN) :: count, dest, tag
                                                                                  5
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  6
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  7
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  9
                                                                                  10
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror) !(_c)
                                                                                  11
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  12
                                                                                  13
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  14
    INTEGER, INTENT(IN) :: dest, tag
                                                                                  15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  16
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
                                                                                  19
             recvcount, recvtype, source, recvtag, comm, status, ierror)
                                                                                  20
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  21
    INTEGER, INTENT(IN) :: sendcount, dest, sendtag, recvcount, source,
                                                                                  22
              recvtag
                                                                                  23
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  24
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  26
    TYPE(MPI_Status) :: status
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
                                                                                  29
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
                                                                                  30
             recvcount, recvtype, source, recvtag, comm, status, ierror)
                                                                                  31
              !(_c)
                                                                                  32
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  33
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  34
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  35
    INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
                                                                                  36
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  38
    TYPE(MPI_Status) :: status
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  40
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
                                                                                  41
             comm, status, ierror)
                                                                                  42
    TYPE(*), DIMENSION(..) :: buf
                                                                                  43
    INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  46
    TYPE(MPI_Status) :: status
                                                                                  47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  48
```

```
1
    MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
\mathbf{2}
                   comm, status, ierror) !(_c)
3
         TYPE(*), DIMENSION(..) :: buf
4
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
5
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
6
         INTEGER, INTENT(IN) :: dest, sendtag, source, recvtag
7
         TYPE(MPI_Comm), INTENT(IN) :: comm
8
         TYPE(MPI_Status) :: status
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
    MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
11
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
12
         INTEGER, INTENT(IN) :: count, dest, tag
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror) !(_c)
18
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         INTEGER, INTENT(IN) :: dest, tag
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
25
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
26
         INTEGER, INTENT(IN) :: count, dest, tag
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         TYPE(MPI_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
33
                   !(_c)
34
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
35
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         INTEGER, INTENT(IN) :: dest, tag
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Request), INTENT(OUT) :: request
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
    MPI_Start(request, ierror)
42
         TYPE(MPI_Request), INTENT(INOUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
    MPI_Startall(count, array_of_requests, ierror)
46
         INTEGER, INTENT(IN) :: count
47
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   2
MPI_Test(request, flag, status, ierror)
                                                                                   3
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                   4
    LOGICAL, INTENT(OUT) :: flag
                                                                                   5
    TYPE(MPI_Status) :: status
                                                                                   6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   7
                                                                                   8
MPI_Test_cancelled(status, flag, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
                                                                                   9
                                                                                  10
    LOGICAL, INTENT(OUT) :: flag
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
                                                                                  13
    INTEGER, INTENT(IN) :: count
                                                                                  14
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  15
    LOGICAL, INTENT(OUT) :: flag
                                                                                  16
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
                                                                                  19
MPI_Testany(count, array_of_requests, index, flag, status, ierror)
                                                                                  20
    INTEGER, INTENT(IN) :: count
                                                                                  21
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  22
    INTEGER, INTENT(OUT) :: index
                                                                                  23
    LOGICAL, INTENT(OUT) :: flag
                                                                                  24
    TYPE(MPI_Status) :: status
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
                                                                                  27
             array_of_statuses, ierror)
                                                                                  28
    INTEGER, INTENT(IN) :: incount
                                                                                  29
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
                                                                                  30
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
                                                                                  31
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Wait(request, status, ierror)
                                                                                  35
    TYPE(MPI_Request), INTENT(INOUT) :: request
                                                                                  36
    TYPE(MPI_Status) :: status
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
                                                                                  39
    INTEGER, INTENT(IN) :: count
                                                                                  40
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  41
    TYPE(MPI_Status) :: array_of_statuses(*)
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
                                                                                  44
MPI_Waitany(count, array_of_requests, index, status, ierror)
                                                                                  45
    INTEGER, INTENT(IN) :: count
                                                                                  46
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
                                                                                  47
    INTEGER, INTENT(OUT) :: index
                                                                                  48
```

```
1
         TYPE(MPI_Status) :: status
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
4
                   array_of_statuses, ierror)
5
         INTEGER, INTENT(IN) :: incount
6
         TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(incount)
7
         INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
8
         TYPE(MPI_Status) :: array_of_statuses(*)
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
12
     A.4.2 Partitioned Communication Fortran 2008 Bindings
13
    MPI_Parrived(request, partition, flag, ierror)
14
         TYPE(MPI_Request), INTENT(IN) :: request
15
         INTEGER, INTENT(IN) :: partition
16
         LOGICAL, INTENT(OUT) :: flag
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
    MPI_Pready(partition, request, ierror)
20
         INTEGER, INTENT(IN) :: partition
21
         TYPE(MPI_Request), INTENT(IN) :: request
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
    MPI_Pready_list(length, array_of_partitions, request, ierror)
^{24}
         INTEGER, INTENT(IN) :: length, array_of_partitions(length)
25
         TYPE(MPI_Request), INTENT(IN) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Pready_range(partition_low, partition_high, request, ierror)
29
         INTEGER, INTENT(IN) :: partition_low, partition_high
30
         TYPE(MPI_Request), INTENT(IN) :: request
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Precv_init(buf, partitions, count, datatype, source, tag, comm, info,
33
34
                  request, ierror)
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
35
         INTEGER, INTENT(IN) :: partitions, source, tag
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
37
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         TYPE(MPI_Info), INTENT(IN) :: info
40
         TYPE(MPI_Request), INTENT(OUT) :: request
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Psend_init(buf, partitions, count, datatype, dest, tag, comm, info,
44
                  request, ierror)
45
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
46
         INTEGER, INTENT(IN) :: partitions, dest, tag
47
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   3
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
                                                                                   6
                                                                                   7
A.4.3 Datatypes Fortran 2008 Bindings
                                                                                   8
                                                                                  9
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
                                                                                  10
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp
                                                                                  11
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
                                                                                  12
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2
                                                                                  13
                                                                                  14
MPI_Get_address(location, address, ierror)
                                                                                  15
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
                                                                                  16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
MPI_Get_elements(status, datatype, count, ierror)
                                                                                  19
    TYPE(MPI_Status), INTENT(IN) :: status
                                                                                  20
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  21
    INTEGER, INTENT(OUT) :: count
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
                                                                                  24
MPI_Get_elements(status, datatype, count, ierror) !(_c)
                                                                                  25
    TYPE(MPI_Status), INTENT(IN) :: status
                                                                                  26
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  27
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
                                                                                  28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  29
MPI_Get_elements_x(status, datatype, count, ierror)
                                                                                  30
    TYPE(MPI_Status), INTENT(IN) :: status
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  32
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: count
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  35
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
                                                                                  36
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                  37
    INTEGER, INTENT(IN) :: incount, outsize
                                                                                  38
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  39
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                  40
    INTEGER, INTENT(INOUT) :: position
                                                                                  41
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
                                                                                  44
              !(_c)
                                                                                  45
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                  46
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize
                                                                                  47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  48
```

```
1
         TYPE(*), DIMENSION(..) :: outbuf
2
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
6
                   position, ierror)
7
         CHARACTER(LEN=*), INTENT(IN) :: datarep
8
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
9
         INTEGER, INTENT(IN) :: incount
10
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
11
         TYPE(*), DIMENSION(..) :: outbuf
12
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
13
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
17
                  position, ierror) !(_c)
18
         CHARACTER(LEN=*), INTENT(IN) :: datarep
19
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
20
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount, outsize
21
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
22
         TYPE(*), DIMENSION(..) :: outbuf
23
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
^{24}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
26
         CHARACTER(LEN=*), INTENT(IN) :: datarep
27
         INTEGER, INTENT(IN) :: incount
28
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
29
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Pack_external_size(datarep, incount, datatype, size, ierror) !(_c)
33
         CHARACTER(LEN=*), INTENT(IN) :: datarep
34
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Pack_size(incount, datatype, comm, size, ierror)
39
         INTEGER, INTENT(IN) :: incount
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         INTEGER, INTENT(OUT) :: size
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Pack_size(incount, datatype, comm, size, ierror) !(_c)
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: incount
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  1
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
                                                                                  2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
MPI_Type_commit(datatype, ierror)
                                                                                  5
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
                                                                                  6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  7
                                                                                  8
MPI_Type_contiguous(count, oldtype, newtype, ierror)
                                                                                  9
    INTEGER, INTENT(IN) :: count
                                                                                  10
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  11
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
                                                                                  13
MPI_Type_contiguous(count, oldtype, newtype, ierror) !(_c)
                                                                                  14
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  15
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  16
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
                                                                                  19
MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
                                                                                  20
              array_of_distribs, array_of_dargs, array_of_psizes, order,
                                                                                  21
              oldtype, newtype, ierror)
                                                                                  22
    INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
                                                                                  23
              array_of_distribs(ndims), array_of_dargs(ndims),
                                                                                  24
              array_of_psizes(ndims), order
                                                                                  25
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  26
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
                                                                                  29
              array_of_distribs, array_of_dargs, array_of_psizes, order,
                                                                                  30
              oldtype, newtype, ierror) !(_c)
                                                                                  31
    INTEGER, INTENT(IN) :: size, rank, ndims, array_of_distribs(ndims),
                                                                                  32
              array_of_dargs(ndims), array_of_psizes(ndims), order
                                                                                  33
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_gsizes(ndims)
                                                                                  34
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  35
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  37
                                                                                  38
MPI_Type_create_hindexed(count, array_of_blocklengths,
                                                                                  39
              array_of_displacements, oldtype, newtype, ierror)
                                                                                  40
    INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
                                                                                  41
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                                  42
              array_of_displacements(count)
                                                                                  43
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  44
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
MPI_Type_create_hindexed(count, array_of_blocklengths,
                                                                                  47
              array_of_displacements, oldtype, newtype, ierror) !(_c)
                                                                                  48
```

```
1
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
2
                   array_of_blocklengths(count), array_of_displacements(count)
3
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
4
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
7
                  oldtype, newtype, ierror)
8
         INTEGER, INTENT(IN) :: count, blocklength
9
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
10
                   array_of_displacements(count)
11
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
12
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
16
                   oldtype, newtype, ierror) !(_c)
17
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,
18
                   array_of_displacements(count)
19
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
20
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
23
                  ierror)
24
         INTEGER, INTENT(IN) :: count, blocklength
25
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
26
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
27
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
^{31}
                  ierror) !(_c)
32
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride
33
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
34
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
37
                  oldtype, newtype, ierror)
38
         INTEGER, INTENT(IN) :: count, blocklength,
39
                   array_of_displacements(count)
40
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
41
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
45
                  oldtype, newtype, ierror) !(_c)
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength,
47
                   array_of_displacements(count)
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  2
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
                                                                                  5
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  6
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
                                                                                  7
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  9
                                                                                  10
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror) !(_c)
                                                                                  11
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: lb, extent
                                                                                  12
                                                                                  13
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                  15
MPI_Type_create_struct(count, array_of_blocklengths,
                                                                                  16
              array_of_displacements, array_of_types, newtype, ierror)
                                                                                  17
    INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
                                                                                  18
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
                                                                                  19
              array_of_displacements(count)
                                                                                  20
    TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
                                                                                  21
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
                                                                                  24
MPI_Type_create_struct(count, array_of_blocklengths,
                                                                                  25
              array_of_displacements, array_of_types, newtype, ierror) !(_c)
                                                                                  26
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,
                                                                                  27
              array_of_blocklengths(count), array_of_displacements(count)
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
                                                                                  29
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  31
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                                                                                  32
             array_of_starts, order, oldtype, newtype, ierror)
                                                                                  33
    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
                                                                                  34
              array_of_subsizes(ndims), array_of_starts(ndims), order
                                                                                  35
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  36
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
                                                                                  39
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
                                                                                  40
              array_of_starts, order, oldtype, newtype, ierror) !(_c)
                                                                                  41
    INTEGER, INTENT(IN) :: ndims, order
                                                                                  42
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: array_of_sizes(ndims),
                                                                                  43
              array_of_subsizes(ndims), array_of_starts(ndims)
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
                                                                                  45
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
                                                                                  46
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  47
MPI_Type_dup(oldtype, newtype, ierror)
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
2
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Type_free(datatype, ierror)
5
         TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
9
                   array_of_integers, array_of_addresses, array_of_datatypes,
10
                   ierror)
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         INTEGER, INTENT(IN) :: max_integers, max_addresses, max_datatypes
13
         INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
14
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
15
                   array_of_addresses(max_addresses)
16
         TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Type_get_contents(datatype, max_integers, max_addresses,
19
                   max_large_counts, max_datatypes, array_of_integers,
20
                   array_of_addresses, array_of_large_counts, array_of_datatypes,
21
                   ierror) !(_c)
22
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
23
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: max_integers,
24
                   max_addresses, max_large_counts, max_datatypes
25
         INTEGER, INTENT(OUT) :: array_of_integers(max_integers)
26
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
27
                   array_of_addresses(max_addresses)
28
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::
29
                   array_of_large_counts(max_large_counts)
30
         TYPE(MPI_Datatype), INTENT(OUT) :: array_of_datatypes(max_datatypes)
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
34
                   combiner, ierror)
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
37
                    combiner
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Type_get_envelope(datatype, num_integers, num_addresses,
40
                   num_large_counts, num_datatypes, combiner, ierror) !(_c)
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: num_integers,
43
                   num_addresses, num_large_counts, num_datatypes
44
         INTEGER, INTENT(OUT) :: combiner
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Type_get_extent(datatype, lb, extent, ierror)
48
```

TYPE(MPI_Datatype), INTENT(IN) :: datatype	1
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: lb, extent	2
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	3 4
MPI_Type_get_extent(datatype, lb, extent, ierror) !(_c)	4 5
TYPE(MPI_Datatype), INTENT(IN) :: datatype	6
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
<pre>MPI_Type_get_extent_x(datatype, lb, extent, ierror)</pre>	9
TYPE(MPI_Datatype), INTENT(IN) :: datatype	10
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: lb, extent	11
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	12
	13
<pre>MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)     TYPE(MPI_Datatype), INTENT(IN) :: datatype</pre>	14
INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
	17
<pre>MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror) !(_c)</pre>	18 19
TYPE(MPI_Datatype), INTENT(IN) :: datatype	20
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	22
<pre>MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)</pre>	23
TYPE(MPI_Datatype), INTENT(IN) :: datatype	24
INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent	25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	26
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,	27
oldtype, newtype, ierror)	28
INTEGER, INTENT(IN) :: count, array_of_blocklengths(count),	29
array_of_displacements(count)	30
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	31 32
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,	35
oldtype, newtype, ierror) !(_c)	36
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count,	37
array_of_blocklengths(count), array_of_displacements(count)	38
TYPE(MPI_Datatype), INTENT(IN) :: oldtype	39
TYPE(MPI_Datatype), INTENT(OUT) :: newtype	40
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	41
MPI_Type_size(datatype, size, ierror)	42
TYPE(MPI_Datatype), INTENT(IN) :: datatype	43
INTEGER, INTENT(OUT) :: size	44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	45
<pre>MPI_Type_size(datatype, size, ierror) !(_c)</pre>	46 47
TYPE(MPI_Datatype), INTENT(IN) :: datatype	48
	10

```
1
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Type_size_x(datatype, size, ierror)
4
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
    MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
9
         INTEGER, INTENT(IN) :: count, blocklength, stride
10
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
11
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
    MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror) !(_c)
14
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count, blocklength, stride
15
         TYPE(MPI_Datatype), INTENT(IN) :: oldtype
16
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
19
    MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
20
                   ierror)
21
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
22
         INTEGER, INTENT(IN) :: insize, outcount
23
         INTEGER, INTENT(INOUT) :: position
^{24}
         TYPE(*), DIMENSION(..) :: outbuf
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         TYPE(MPI_Comm), INTENT(IN) :: comm
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
29
                   ierror) !(_c)
30
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
31
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
32
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
33
         TYPE(*), DIMENSION(..) :: outbuf
34
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
39
                  datatype, ierror)
40
         CHARACTER(LEN=*), INTENT(IN) :: datarep
41
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: insize
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
44
         TYPE(*), DIMENSION(...) :: outbuf
45
         INTEGER, INTENT(IN) :: outcount
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outcount,
                                                                                   2
              datatype, ierror) !(_c)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
                                                                                   4
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: insize, outcount
                                                                                  5
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(INOUT) :: position
                                                                                  6
                                                                                  7
    TYPE(*), DIMENSION(..) :: outbuf
                                                                                   8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  9
                                                                                  10
                                                                                  11
A.4.4 Collective Communication Fortran 2008 Bindings
                                                                                  12
                                                                                  13
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  14
              comm, ierror)
                                                                                  15
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  16
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  17
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  18
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  19
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  22
              comm, ierror) !(_c)
                                                                                  23
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  24
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  25
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  26
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  27
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  29
                                                                                  30
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  31
             recvtype, comm, info, request, ierror)
                                                                                  32
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  33
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  34
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  35
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  37
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  38
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  40
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  41
              recvtype, comm, info, request, ierror) !(_c)
                                                                                  42
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  43
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  48
```

939

```
1
         TYPE(MPI_Request), INTENT(OUT) :: request
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
4
                  recvtype, comm, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
6
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
7
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
8
         TYPE(*), DIMENSION(...) :: recvbuf
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
13
                  recvtype, comm, ierror) !(_c)
14
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
15
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
16
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
17
         TYPE(*), DIMENSION(...) :: recvbuf
18
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
22
                  displs, recvtype, comm, info, request, ierror)
23
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
         INTEGER, INTENT(IN) :: sendcount
25
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
28
         TYPE(MPI_Comm), INTENT(IN) :: comm
29
         TYPE(MPI_Info), INTENT(IN) :: info
30
         TYPE(MPI_Request), INTENT(OUT) :: request
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
33
     MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
34
                  displs, recvtype, comm, info, request, ierror) !(_c)
35
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
36
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
37
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
39
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
40
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Info), INTENT(IN) :: info
43
         TYPE(MPI_Request), INTENT(OUT) :: request
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
     MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
46
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
47
         TYPE(*), DIMENSION(...) :: recvbuf
48
```

```
1
    INTEGER, INTENT(IN) :: count
                                                                                   2
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  4
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
                                                                                   6
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
                                                                                  7
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                   8
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  9
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  10
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  11
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  12
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                  15
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
                                                                                  16
              request, ierror)
                                                                                  17
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  18
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  19
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  20
                                                                                  21
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  23
                                                                                  24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
                                                                                  27
             request, ierror) !(_c)
                                                                                  28
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  29
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  30
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  32
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  33
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  34
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  35
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  37
                                                                                  38
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  39
              comm, ierror)
                                                                                  40
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  41
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  43
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  47
              comm, ierror) !(_c)
                                                                                  48
```

```
1
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
2
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
3
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
4
         TYPE(*), DIMENSION(..) :: recvbuf
5
         TYPE(MPI_Comm), INTENT(IN) :: comm
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
8
                  recvtype, comm, info, request, ierror)
9
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
10
         INTEGER, INTENT(IN) :: sendcount, recvcount
11
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Info), INTENT(IN) :: info
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
19
                   recvtype, comm, info, request, ierror) !(_c)
20
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
21
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
22
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
^{24}
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Info), INTENT(IN) :: info
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
29
                   rdispls, recvtype, comm, ierror)
30
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
31
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
32
                   rdispls(*)
33
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
         TYPE(*), DIMENSION(..) :: recvbuf
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
39
                   rdispls, recvtype, comm, ierror) !(_c)
40
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
42
                   recvcounts(*)
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
44
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
45
         TYPE(*), DIMENSION(...) :: recvbuf
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  2
             recvcounts, rdispls, recvtype, comm, info, request, ierror)
                                                                                   3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  4
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
                                                                                  5
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  6
                                                                                  7
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   8
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  9
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  11
                                                                                  12
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
                                                                                  13
             recvcounts, rdispls, recvtype, comm, info, request, ierror)
                                                                                  14
              !(_c)
                                                                                  15
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  16
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                  17
              sendcounts(*), recvcounts(*)
                                                                                  18
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  19
              rdispls(*)
                                                                                  20
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  23
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
                                                                                  27
MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
                                                                                  28
             rdispls, recvtypes, comm, ierror)
                                                                                  29
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  30
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
                                                                                  31
              rdispls(*)
                                                                                  32
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                  33
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  36
MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
                                                                                  37
             rdispls, recvtypes, comm, ierror) !(_c)
                                                                                  38
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  39
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
                                                                                  40
              recvcounts(*)
                                                                                  41
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                  43
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  44
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
                                                                                  47
```

48

```
1
     MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
2
                   recvcounts, rdispls, recvtypes, comm, info, request, ierror)
3
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
4
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
5
                   recvcounts(*), rdispls(*)
6
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
7
                   recvtypes(*)
8
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         TYPE(MPI_Request), INTENT(OUT) :: request
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
     MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
14
                   recvcounts, rdispls, recvtypes, comm, info, request, ierror)
15
                   !(_c)
16
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
17
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
18
                   sendcounts(*), recvcounts(*)
19
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
20
                   rdispls(*)
21
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
22
                   recvtypes(*)
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
24
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Info), INTENT(IN) :: info
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
29
     MPI_Barrier(comm, ierror)
30
         TYPE(MPI_Comm), INTENT(IN) :: comm
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Barrier_init(comm, info, request, ierror)
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         TYPE(MPI_Info), INTENT(IN) :: info
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Bcast(buffer, count, datatype, root, comm, ierror)
39
         TYPE(*), DIMENSION(..) :: buffer
40
         INTEGER, INTENT(IN) :: count, root
41
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
42
         TYPE(MPI_Comm), INTENT(IN) :: comm
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Bcast(buffer, count, datatype, root, comm, ierror) !(_c)
45
         TYPE(*), DIMENSION(..) :: buffer
46
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   2
                                                                                   3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
                                                                                   5
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                   6
    INTEGER, INTENT(IN) :: count, root
                                                                                   7
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   8
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   9
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
                                                                                  13
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
                                                                                  14
              !(_c)
                                                                                  15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                  16
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  17
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  18
    INTEGER, INTENT(IN) :: root
                                                                                  19
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  20
                                                                                  21
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
                                                                                  24
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  25
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  26
    INTEGER, INTENT(IN) :: count
                                                                                  27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  28
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  31
                                                                                  32
MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
                                                                                  33
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  34
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  35
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  37
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  38
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  40
MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                  41
              ierror)
                                                                                  42
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  43
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  44
    INTEGER, INTENT(IN) :: count
                                                                                  45
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  46
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  48
```

```
1
         TYPE(MPI_Info), INTENT(IN) :: info
2
         TYPE(MPI_Request), INTENT(OUT) :: request
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
5
                   ierror) !(_c)
6
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Op), INTENT(IN) :: op
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Info), INTENT(IN) :: info
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
17
                  root, comm, ierror)
18
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
19
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
         TYPE(*), DIMENSION(...) :: recvbuf
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
25
                   root, comm, ierror) !(_c)
26
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
27
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
28
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
         TYPE(*), DIMENSION(...) :: recvbuf
30
         INTEGER, INTENT(IN) :: root
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
35
                   root, comm, info, request, ierror)
36
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
37
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
38
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
39
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
40
         TYPE(MPI_Comm), INTENT(IN) :: comm
41
         TYPE(MPI_Info), INTENT(IN) :: info
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
45
                   root, comm, info, request, ierror) !(_c)
46
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
47
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  2
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  4
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  5
                                                                                  6
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  9
             recvtype, root, comm, ierror)
                                                                                  10
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  11
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
                                                                                  12
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  13
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  14
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  16
                                                                                  17
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  18
             recvtype, root, comm, ierror) !(_c)
                                                                                  19
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
                                                                                  20
                                                                                  21
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  22
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  23
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                  24
    INTEGER, INTENT(IN) :: root
                                                                                  25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
MPI_Gathery_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  28
             recvtype, root, comm, info, request, ierror)
                                                                                  29
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  30
    INTEGER, INTENT(IN) :: sendcount, root
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  32
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  33
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
                                                                                  34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  35
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  36
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
                                                                                  39
MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
                                                                                  40
             recvtype, root, comm, info, request, ierror) !(_c)
                                                                                  41
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  42
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
                                                                                  43
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  44
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  45
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  46
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                  47
    INTEGER, INTENT(IN) :: root
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Info), INTENT(IN) :: info
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
6
                  comm, request, ierror)
7
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         INTEGER, INTENT(IN) :: sendcount, recvcount
9
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
16
                   comm, request, ierror) !(_c)
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
19
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
21
         TYPE(MPI_Comm), INTENT(IN) :: comm
22
         TYPE(MPI_Request), INTENT(OUT) :: request
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
25
                  recvtype, comm, request, ierror)
26
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
27
         INTEGER, INTENT(IN) :: sendcount
28
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
29
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
30
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
31
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         TYPE(MPI_Request), INTENT(OUT) :: request
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
36
                  recvtype, comm, request, ierror) !(_c)
37
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
38
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
39
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
40
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
42
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Request), INTENT(OUT) :: request
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
47
                   ierror)
48
```

```
TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   1
                                                                                   2
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                   3
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   4
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   5
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   6
                                                                                  7
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   8
                                                                                  9
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
                                                                                  10
              ierror) !(_c)
                                                                                  11
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  12
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  13
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  14
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  15
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  16
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  17
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  19
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  20
                                                                                  21
              comm, request, ierror)
                                                                                  22
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  23
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  24
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recytype
                                                                                  25
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  26
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  27
                                                                                  28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  29
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  30
              comm, request, ierror) !(_c)
                                                                                  31
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  32
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  33
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  34
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  38
                                                                                  39
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
                                                                                  40
              rdispls, recvtype, comm, request, ierror)
                                                                                  41
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  42
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
                                                                                  43
              recvcounts(*), rdispls(*)
                                                                                  44
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  45
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  46
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  47
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
3
                   rdispls, recvtype, comm, request, ierror) !(_c)
4
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
6
                   sendcounts(*), recvcounts(*)
7
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
8
                   rdispls(*)
9
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         TYPE(MPI_Comm), INTENT(IN) :: comm
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
16
                   recvcounts, rdispls, recvtypes, comm, request, ierror)
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
19
                   recvcounts(*), rdispls(*)
20
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
21
                   recvtypes(*)
22
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         TYPE(MPI_Request), INTENT(OUT) :: request
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
     MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
27
                   recvcounts, rdispls, recvtypes, comm, request, ierror) !(_c)
28
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
29
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
30
                   sendcounts(*), recvcounts(*)
31
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
32
                   rdispls(*)
33
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
34
                   recvtypes(*)
35
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
36
         TYPE(MPI_Comm), INTENT(IN) :: comm
37
         TYPE(MPI_Request), INTENT(OUT) :: request
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Ibarrier(comm, request, ierror)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm
42
         TYPE(MPI_Request), INTENT(OUT) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
     MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
45
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
46
         INTEGER, INTENT(IN) :: count, root
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   1
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror) !(_c)
                                                                                   5
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buffer
                                                                                   6
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                   7
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   8
    INTEGER, INTENT(IN) :: root
                                                                                   9
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
                                                                                  13
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                  14
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  16
    INTEGER, INTENT(IN) :: count
                                                                                  17
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  18
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  19
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
                                                                                  22
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                  23
              !( c)
                                                                                  24
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  25
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
                                                                                  26
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  28
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  30
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
                                                                                  33
MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  34
              root, comm, request, ierror)
                                                                                  35
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  36
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  38
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  40
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  42
MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  43
              root, comm, request, ierror) !(_c)
                                                                                  44
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  45
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  48
```

```
1
         INTEGER, INTENT(IN) :: root
2
         TYPE(MPI_Comm), INTENT(IN) :: comm
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
    MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
6
                  recvtype, root, comm, request, ierror)
7
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         INTEGER, INTENT(IN) :: sendcount, root
9
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Request), INTENT(OUT) :: request
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
    MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
17
                  recvtype, root, comm, request, ierror) !(_c)
18
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
20
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
21
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
22
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
23
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
^{24}
         INTEGER, INTENT(IN) :: root
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
     MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
29
                   ierror)
30
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
31
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
32
         INTEGER, INTENT(IN) :: count, root
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Op), INTENT(IN) :: op
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         TYPE(MPI_Request), INTENT(OUT) :: request
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
40
                   ierror) !(_c)
41
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
42
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
43
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
44
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
         TYPE(MPI_Op), INTENT(IN) :: op
46
         INTEGER, INTENT(IN) :: root
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   2
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                   4
              request, ierror)
                                                                                   5
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  6
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  7
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                   8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  9
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  10
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  13
                                                                                  14
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                  15
             request, ierror) !(_c)
                                                                                  16
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  17
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  18
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  20
                                                                                  21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  22
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                  25
             request, ierror)
                                                                                  26
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  27
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  28
    INTEGER, INTENT(IN) :: recvcount
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  30
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  32
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  35
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                  36
              request, ierror) !(_c)
                                                                                  37
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  38
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  39
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  41
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  43
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  45
MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
                                                                                  46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  48
```

```
1
         INTEGER, INTENT(IN) :: count
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Op), INTENT(IN) :: op
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
8
                   !(_c)
9
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
12
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
13
         TYPE(MPI_Op), INTENT(IN) :: op
14
         TYPE(MPI_Comm), INTENT(IN) :: comm
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
19
                   root, comm, request, ierror)
20
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
21
         INTEGER, INTENT(IN) :: sendcount, recvcount, root
22
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
^{24}
         TYPE(MPI_Comm), INTENT(IN) :: comm
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
28
                   root, comm, request, ierror) !(_c)
29
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
31
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
32
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
33
         INTEGER, INTENT(IN) :: root
34
         TYPE(MPI_Comm), INTENT(IN) :: comm
35
         TYPE(MPI_Request), INTENT(OUT) :: request
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
39
                   recvtype, root, comm, request, ierror)
40
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
41
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
42
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
43
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
44
         INTEGER, INTENT(IN) :: recvcount, root
45
         TYPE(MPI_Comm), INTENT(IN) :: comm
46
         TYPE(MPI_Request), INTENT(OUT) :: request
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
              recvtype, root, comm, request, ierror) !(_c)
                                                                                   2
                                                                                   3
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
                                                                                   4
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                   5
                                                                                   6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   7
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                   8
    INTEGER, INTENT(IN) :: root
                                                                                   9
                                                                                   10
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
                                                                                  13
MPI_Op_commutative(op, commute, ierror)
                                                                                  14
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   15
    LOGICAL, INTENT(OUT) :: commute
                                                                                   16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   17
                                                                                   18
MPI_Op_create(user_fn, commute, op, ierror)
                                                                                  19
    PROCEDURE(MPI_User_function) :: user_fn
    LOGICAL, INTENT(IN) :: commute
                                                                                  20
                                                                                  21
    TYPE(MPI_Op), INTENT(OUT) :: op
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
MPI_Op_create_c(user_fn, commute, op, ierror) !(_c)
                                                                                  24
    PROCEDURE(MPI_User_function_c) :: user_fn
                                                                                  25
    LOGICAL, INTENT(IN) :: commute
                                                                                  26
    TYPE(MPI_Op), INTENT(OUT) :: op
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
                                                                                  29
MPI_Op_free(op, ierror)
                                                                                  30
    TYPE(MPI_Op), INTENT(INOUT) :: op
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
                                                                                  33
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  34
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  35
    INTEGER, INTENT(IN) :: count, root
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  37
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  38
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   40
                                                                                  41
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror) !(_c)
                                                                                  42
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  43
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  44
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  45
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   46
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   47
    INTEGER, INTENT(IN) :: root
                                                                                   48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
4
                   request, ierror)
5
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
6
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
7
         INTEGER, INTENT(IN) :: count, root
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         TYPE(MPI_Op), INTENT(IN) :: op
10
         TYPE(MPI_Comm), INTENT(IN) :: comm
11
         TYPE(MPI_Info), INTENT(IN) :: info
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
16
                   request, ierror) !(_c)
17
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
18
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
19
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
20
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
21
         TYPE(MPI_Op), INTENT(IN) :: op
22
         INTEGER, INTENT(IN) :: root
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
^{24}
         TYPE(MPI_Info), INTENT(IN) :: info
25
         TYPE(MPI_Request), INTENT(OUT) :: request
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
28
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
29
         TYPE(*), DIMENSION(..) :: inoutbuf
30
         INTEGER, INTENT(IN) :: count
31
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
32
         TYPE(MPI_Op), INTENT(IN) :: op
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
35
     MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror) !(_c)
36
         TYPE(*), DIMENSION(...), INTENT(IN) :: inbuf
37
         TYPE(*), DIMENSION(..) :: inoutbuf
38
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
39
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
40
         TYPE(MPI_Op), INTENT(IN) :: op
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
43
                   ierror)
44
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
45
         TYPE(*), DIMENSION(...) :: recvbuf
46
         INTEGER, INTENT(IN) :: recvcounts(*)
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
                                                                                   5
              ierror) !(_c)
                                                                                   6
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                   7
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                   8
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcounts(*)
                                                                                   9
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   10
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  13
                                                                                  14
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                  15
             ierror)
                                                                                  16
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                   17
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  18
    INTEGER, INTENT(IN) :: recvcount
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  20
                                                                                  21
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
                                                                                  24
              ierror) !(_c)
                                                                                  25
    TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
                                                                                  26
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  27
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  29
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  30
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
                                                                                  33
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
                                                                                  34
              comm, info, request, ierror)
                                                                                  35
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  36
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  37
    INTEGER, INTENT(IN) :: recvcount
                                                                                  38
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  39
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                   40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  41
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  42
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
                                                                                  45
              comm, info, request, ierror) !(_c)
                                                                                  46
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  48
```

```
1
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Op), INTENT(IN) :: op
4
         TYPE(MPI_Comm), INTENT(IN) :: comm
5
         TYPE(MPI_Info), INTENT(IN) :: info
6
         TYPE(MPI_Request), INTENT(OUT) :: request
7
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
8
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
9
                   info, request, ierror)
10
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
11
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
12
         INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
13
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
14
         TYPE(MPI_Op), INTENT(IN) :: op
15
         TYPE(MPI_Comm), INTENT(IN) :: comm
16
         TYPE(MPI_Info), INTENT(IN) :: info
17
         TYPE(MPI_Request), INTENT(OUT) :: request
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
20
     MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
21
                   info, request, ierror) !(_c)
22
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
^{24}
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         TYPE(MPI_Op), INTENT(IN) :: op
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(MPI_Info), INTENT(IN) :: info
29
         TYPE(MPI_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
32
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
33
         TYPE(*), DIMENSION(..) :: recvbuf
34
         INTEGER, INTENT(IN) :: count
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         TYPE(MPI_Op), INTENT(IN) :: op
37
         TYPE(MPI_Comm), INTENT(IN) :: comm
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror) !(_c)
41
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
42
         TYPE(*), DIMENSION(..) :: recvbuf
43
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
44
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
45
         TYPE(MPI_Op), INTENT(IN) :: op
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                   2
              ierror)
                                                                                   3
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  4
    INTEGER, INTENT(IN) :: count
                                                                                   5
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                   6
                                                                                  7
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   8
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  9
                                                                                  10
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  11
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  12
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
                                                                                  13
              ierror) !(_c)
                                                                                  14
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  15
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  16
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  17
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  18
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  19
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  20
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  21
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
                                                                                  24
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  25
             root, comm, ierror)
                                                                                  26
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  27
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  28
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  29
    TYPE(*), DIMENSION(...) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  30
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                                                                                  33
             root, comm, ierror) !(_c)
                                                                                  34
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  35
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  36
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  37
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  38
    INTEGER, INTENT(IN) :: root
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  43
              recvtype, root, comm, info, request, ierror)
                                                                                  44
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  45
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Info), INTENT(IN) :: info
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
6
                   recvtype, root, comm, info, request, ierror) !(_c)
7
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
9
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
10
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
11
         INTEGER, INTENT(IN) :: root
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         TYPE(MPI_Info), INTENT(IN) :: info
14
         TYPE(MPI_Request), INTENT(OUT) :: request
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
18
                   recvtype, root, comm, ierror)
19
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
20
         INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
21
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
22
         TYPE(*), DIMENSION(...) :: recvbuf
23
         TYPE(MPI_Comm), INTENT(IN) :: comm
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
26
                   recvtype, root, comm, ierror) !(_c)
27
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
28
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*), recvcount
29
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
30
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
31
         TYPE(*), DIMENSION(..) :: recvbuf
32
         INTEGER, INTENT(IN) :: root
33
         TYPE(MPI_Comm), INTENT(IN) :: comm
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Scattery_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
37
                   recvcount, recvtype, root, comm, info, request, ierror)
38
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
39
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
40
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
41
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
42
         INTEGER, INTENT(IN) :: recvcount, root
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         TYPE(MPI_Info), INTENT(IN) :: info
45
         TYPE(MPI_Request), INTENT(OUT) :: request
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
                                                                                   2
              recvcount, recvtype, root, comm, info, request, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  4
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: sendcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: displs(*)
                                                                                  5
                                                                                   6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                   7
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: recvcount
                                                                                   8
    INTEGER, INTENT(IN) :: root
                                                                                   9
                                                                                  10
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  13
                                                                                  14
                                                                                  15
A.4.5 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings
                                                                                  16
                                                                                  17
MPI_Comm_compare(comm1, comm2, result, ierror)
                                                                                  18
    TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
                                                                                  19
    INTEGER, INTENT(OUT) :: result
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
MPI_Comm_create(comm, group, newcomm, ierror)
                                                                                  22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  23
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                  24
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
                                                                                  27
MPI_Comm_create_from_group(group, stringtag, info, errhandler, newcomm,
                                                                                  28
              ierror)
                                                                                  29
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                  30
    CHARACTER(LEN=*), INTENT(IN) :: stringtag
                                                                                  31
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  32
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
                                                                                  33
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
                                                                                  36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  37
    TYPE(MPI_Group), INTENT(IN) :: group
                                                                                  38
    INTEGER, INTENT(IN) :: tag
                                                                                  39
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
                                                                                  43
              extra_state, ierror)
                                                                                  44
    PROCEDURE(MPI_Comm_copy_attr_function) :: comm_copy_attr_fn
                                                                                  45
    PROCEDURE(MPI_Comm_delete_attr_function) :: comm_delete_attr_fn
                                                                                  46
    INTEGER, INTENT(OUT) :: comm_keyval
                                                                                  47
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Comm_delete_attr(comm, comm_keyval, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         INTEGER, INTENT(IN) :: comm_keyval
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_Comm_dup(comm, newcomm, ierror)
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
     MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
12
                   attribute_val_out, flag, ierror)
13
         TYPE(MPI_Comm) :: oldcomm
14
         INTEGER :: comm_keyval, ierror
15
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
16
                    attribute_val_out
17
         LOGICAL :: flag
18
19
     MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         TYPE(MPI_Info), INTENT(IN) :: info
22
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
24
     MPI_Comm_free(comm, ierror)
25
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Comm_free_keyval(comm_keyval, ierror)
29
         INTEGER, INTENT(INOUT) :: comm_keyval
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
     MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         INTEGER, INTENT(IN) :: comm_keyval
34
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
35
         LOGICAL, INTENT(OUT) :: flag
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Comm_get_info(comm, info_used, ierror)
39
         TYPE(MPI_Comm), INTENT(IN) :: comm
40
         TYPE(MPI_Info), INTENT(OUT) :: info_used
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
43
         TYPE(MPI_Comm), INTENT(IN) :: comm
44
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
45
         INTEGER, INTENT(OUT) :: resultlen
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
48
```

```
1
MPI_Comm_group(comm, group, ierror)
                                                                                   2
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   3
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
                                                                                   5
MPI_Comm_idup(comm, newcomm, request, ierror)
                                                                                   6
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   7
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
                                                                                   8
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                   9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   10
                                                                                  11
MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  12
                                                                                  13
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  14
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
                                                                                  15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  17
MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
                                                                                  18
              attribute_val_out, flag, ierror)
                                                                                  19
    TYPE(MPI_Comm) :: oldcomm
                                                                                  20
    INTEGER :: comm_keyval, ierror
                                                                                  21
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                  22
              attribute_val_out
                                                                                  23
    LOGICAL :: flag
                                                                                  24
                                                                                  25
MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
                                                                                  26
              ierror)
                                                                                  27
    TYPE(MPI_Comm) :: comm
                                                                                  28
    INTEGER :: comm_keyval, ierror
                                                                                  29
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                  30
MPI_Comm_rank(comm, rank, ierror)
                                                                                  31
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  32
    INTEGER, INTENT(OUT) :: rank
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  35
MPI_Comm_remote_group(comm, group, ierror)
                                                                                  36
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  37
    TYPE(MPI_Group), INTENT(OUT) :: group
                                                                                  38
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  39
MPI_Comm_remote_size(comm, size, ierror)
                                                                                   40
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  41
    INTEGER, INTENT(OUT) :: size
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
                                                                                  44
MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
                                                                                  45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  46
    INTEGER, INTENT(IN) :: comm_keyval
                                                                                  47
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Comm_set_info(comm, info, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Info), INTENT(IN) :: info
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_Comm_set_name(comm, comm_name, ierror)
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         CHARACTER(LEN=*), INTENT(IN) :: comm_name
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
    MPI_Comm_size(comm, size, ierror)
12
         TYPE(MPI_Comm), INTENT(IN) :: comm
13
         INTEGER, INTENT(OUT) :: size
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
    MPI_Comm_split(comm, color, key, newcomm, ierror)
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         INTEGER, INTENT(IN) :: color, key
19
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, INTENT(IN) :: split_type, key
24
         TYPE(MPI_Info), INTENT(IN) :: info
25
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Comm_test_inter(comm, flag, ierror)
29
         TYPE(MPI_Comm), INTENT(IN) :: comm
30
         LOGICAL, INTENT(OUT) :: flag
31
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
32
     MPI_Group_compare(group1, group2, result, ierror)
33
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
34
         INTEGER, INTENT(OUT) :: result
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
37
     MPI_Group_difference(group1, group2, newgroup, ierror)
38
         TYPE(MPI_Group), INTENT(IN) :: group1, group2
39
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     MPI_Group_excl(group, n, ranks, newgroup, ierror)
42
         TYPE(MPI_Group), INTENT(IN) :: group
43
         INTEGER, INTENT(IN) :: n, ranks(n)
44
         TYPE(MPI_Group), INTENT(OUT) :: newgroup
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Group_free(group, ierror)
48
```

TYPE(MPI_Group), INTENT(INOUT) :: group INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1 $2$
MPI_Group_from_session_pset(session, pset_name, newgroup, ierror)	3 4
TYPE(MPI_Session), INTENT(IN) :: session CHARACTER(LEN=*), INTENT(IN) :: pset_name	5 6
TYPE(MPI_Group), INTENT(OUT) :: newgroup	7
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	8
MPI_Group_incl(group, n, ranks, newgroup, ierror)	9 10
TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(IN) :: n, ranks(n)	11
TYPE(MPI_Group), INTENT(OUT) :: newgroup	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
<pre>MPI_Group_intersection(group1, group2, newgroup, ierror)</pre>	14 15
TYPE(MPI_Group), INTENT(IN) :: group1, group2	16
TYPE(MPI_Group), INTENT(OUT) :: newgroup INTEGER, OPTIONAL, INTENT(OUT) :: ierror	17
	18 19
<pre>MPI_Group_range_excl(group, n, ranges, newgroup, ierror)     TYPE(MPI_Group), INTENT(IN) :: group</pre>	20
INTEGER, INTENT(IN) :: n, ranges(3, n)	21
TYPE(MPI_Group), INTENT(OUT) :: newgroup	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23 24
<pre>MPI_Group_range_incl(group, n, ranges, newgroup, ierror)</pre>	25
TYPE(MPI_Group), INTENT(IN) :: group	26
<pre>INTEGER, INTENT(IN) :: n, ranges(3, n) TYPE(MPI_Group), INTENT(OUT) :: newgroup</pre>	27 28
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28 29
<pre>MPI_Group_rank(group, rank, ierror)</pre>	30
TYPE(MPI_Group), INTENT(IN) :: group	31
INTEGER, INTENT(OUT) :: rank	32 33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
MPI_Group_size(group, size, ierror)	35
TYPE(MPI_Group), INTENT(IN) :: group INTEGER, INTENT(OUT) :: size	36
INTEGER, INTENT(001) SIZE INTEGER, OPTIONAL, INTENT(OUT) :: ierror	37 38
<pre>MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)</pre>	39
TYPE(MPI_Group), INTENT(IN) :: group1, group2	40
INTEGER, INTENT(IN) :: n, ranks1(n)	41 42
INTEGER, INTENT(OUT) :: ranks2(n)	42 43
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44
MPI_Group_union(group1, group2, newgroup, ierror)	45
TYPE(MPI_Group), INTENT(IN) :: group1, group2 TYPE(MPI_Group), INTENT(OUT) :: newgroup	46 47
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	48

```
1
     MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
2
                  tag, newintercomm, ierror)
3
         TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
4
         INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
5
         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Intercomm_create_from_groups(local_group, local_leader, remote_group,
8
                  remote_leader, stringtag, info, errhandler, newintercomm,
9
                   ierror)
10
         TYPE(MPI_Group), INTENT(IN) :: local_group, remote_group
11
         INTEGER, INTENT(IN) :: local_leader, remote_leader
12
         CHARACTER(LEN=*), INTENT(IN) :: stringtag
13
         TYPE(MPI_Info), INTENT(IN) :: info
14
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
15
         TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
19
         TYPE(MPI_Comm), INTENT(IN) :: intercomm
20
         LOGICAL, INTENT(IN) :: high
21
         TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
24
                   extra_state, ierror)
25
         PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
26
         PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
27
         INTEGER, INTENT(OUT) :: type_keyval
28
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Type_delete_attr(datatype, type_keyval, ierror)
32
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
33
         INTEGER, INTENT(IN) :: type_keyval
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
36
                   attribute_val_out, flag, ierror)
37
         TYPE(MPI_Datatype) :: oldtype
38
         INTEGER :: type_keyval, ierror
39
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
40
                   attribute_val_out
41
         LOGICAL :: flag
42
43
     MPI_Type_free_keyval(type_keyval, ierror)
44
         INTEGER, INTENT(INOUT) :: type_keyval
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
     MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
47
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
48
```

```
1
    INTEGER, INTENT(IN) :: type_keyval
                                                                                  2
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
                                                                                  3
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
                                                                                  5
MPI_Type_get_name(datatype, type_name, resultlen, ierror)
                                                                                  6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  7
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
                                                                                  8
    INTEGER, INTENT(OUT) :: resultlen
                                                                                  9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
                                                                                  11
MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
                                                                                  12
                                                                                  13
    TYPE(MPI_Datatype) :: oldtype
                                                                                  14
    INTEGER :: type_keyval, ierror
                                                                                  15
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
                                                                                  16
              attribute_val_out
                                                                                  17
    LOGICAL :: flag
                                                                                  18
MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
                                                                                  19
              ierror)
                                                                                  20
    TYPE(MPI_Datatype) :: datatype
                                                                                  21
    INTEGER :: type_keyval
                                                                                  22
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
                                                                                  23
    INTEGER, INTENT(OUT) :: ierror
                                                                                  24
                                                                                  25
MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
                                                                                  26
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  27
    INTEGER, INTENT(IN) :: type_keyval
                                                                                  28
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
                                                                                  29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  30
MPI_Type_set_name(datatype, type_name, ierror)
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  32
    CHARACTER(LEN=*), INTENT(IN) :: type_name
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
                                                                                  35
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
                                                                                  36
              extra_state, ierror)
                                                                                  37
    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
                                                                                  38
    PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
                                                                                  39
    INTEGER, INTENT(OUT) :: win_keyval
                                                                                  40
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                  41
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  42
MPI_Win_delete_attr(win, win_keyval, ierror)
                                                                                  43
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  44
    INTEGER, INTENT(IN) :: win_keyval
                                                                                  45
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  46
                                                                                  47
                                                                                  48
```

```
1
     MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
\mathbf{2}
                   attribute_val_out, flag, ierror)
3
         TYPE(MPI_Win) :: oldwin
4
         INTEGER :: win_keyval, ierror
\mathbf{5}
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
6
                    attribute_val_out
7
         LOGICAL :: flag
8
     MPI_Win_free_keyval(win_keyval, ierror)
9
         INTEGER, INTENT(INOUT) :: win_keyval
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
13
         TYPE(MPI_Win), INTENT(IN) :: win
14
         INTEGER, INTENT(IN) :: win_keyval
15
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
16
         LOGICAL, INTENT(OUT) :: flag
17
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
18
     MPI_Win_get_name(win, win_name, resultlen, ierror)
19
         TYPE(MPI_Win), INTENT(IN) :: win
20
         CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: win_name
21
         INTEGER, INTENT(OUT) :: resultlen
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
^{24}
     MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
25
                   attribute_val_out, flag, ierror)
26
         TYPE(MPI_Win) :: oldwin
27
         INTEGER :: win_keyval, ierror
28
         INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
29
                   attribute_val_out
30
         LOGICAL :: flag
^{31}
     MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
32
         TYPE(MPI_Win) :: win
33
         INTEGER :: win_keyval, ierror
34
         INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
35
36
     MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
37
         TYPE(MPI_Win), INTENT(IN) :: win
38
         INTEGER, INTENT(IN) :: win_keyval
39
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
     MPI_Win_set_name(win, win_name, ierror)
42
         TYPE(MPI_Win), INTENT(IN) :: win
43
         CHARACTER(LEN=*), INTENT(IN) :: win_name
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
47
48
```

```
A.4.6 Process Topologies Fortran 2008 Bindings
                                                                                   1
                                                                                   2
MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: rank, maxdims
                                                                                   5
    INTEGER, INTENT(OUT) :: coords(maxdims)
                                                                                   6
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   7
MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
                                                                                   9
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
                                                                                   10
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
                                                                                   11
    LOGICAL, INTENT(IN) :: periods(ndims), reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
                                                                                   12
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
                                                                                   15
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   16
    INTEGER, INTENT(IN) :: maxdims
                                                                                   17
    INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
                                                                                   18
    LOGICAL, INTENT(OUT) :: periods(maxdims)
                                                                                   19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   20
                                                                                   21
MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
                                                                                   22
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   23
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
                                                                                   24
    LOGICAL, INTENT(IN) :: periods(ndims)
                                                                                   25
    INTEGER, INTENT(OUT) :: newrank
                                                                                   26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   27
MPI_Cart_rank(comm, coords, rank, ierror)
                                                                                   28
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   29
    INTEGER, INTENT(IN) :: coords(*)
                                                                                   30
    INTEGER, INTENT(OUT) :: rank
                                                                                   31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   32
                                                                                   33
MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
                                                                                   34
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   35
    INTEGER, INTENT(IN) :: direction, disp
                                                                                   36
    INTEGER, INTENT(OUT) :: rank_source, rank_dest
                                                                                   37
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   38
MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
                                                                                   39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   40
    LOGICAL, INTENT(IN) :: remain_dims(*)
                                                                                   41
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
                                                                                   42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   43
                                                                                   44
MPI_Cartdim_get(comm, ndims, ierror)
                                                                                   45
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   46
    INTEGER, INTENT(OUT) :: ndims
                                                                                   47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   48
```

```
1
    MPI_Dims_create(nnodes, ndims, dims, ierror)
\mathbf{2}
         INTEGER, INTENT(IN) :: nnodes, ndims
3
         INTEGER, INTENT(INOUT) :: dims(ndims)
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
6
                   info, reorder, comm_dist_graph, ierror)
7
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
8
         INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
9
                   weights(*)
10
         TYPE(MPI_Info), INTENT(IN) :: info
11
         LOGICAL, INTENT(IN) :: reorder
12
         TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
16
                   outdegree, destinations, destweights, info, reorder,
17
                   comm_dist_graph, ierror)
18
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
19
         INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
20
                   outdegree, destinations(outdegree), destweights(*)
21
         TYPE(MPI_Info), INTENT(IN) :: info
22
         LOGICAL, INTENT(IN) :: reorder
23
         TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
24
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
26
                   maxoutdegree, destinations, destweights, ierror)
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         INTEGER, INTENT(IN) :: maxindegree, maxoutdegree
29
         INTEGER, INTENT(OUT) :: sources(maxindegree),
30
                   destinations(maxoutdegree)
31
         INTEGER :: sourceweights(*), destweights(*)
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         INTEGER, INTENT(OUT) :: indegree, outdegree
37
         LOGICAL, INTENT(OUT) :: weighted
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
     MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
40
                   ierror)
41
         TYPE(MPI_Comm), INTENT(IN) :: comm_old
42
         INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
43
         LOGICAL, INTENT(IN) :: reorder
44
         TYPE(MPI_Comm), INTENT(OUT) :: comm_graph
45
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
46
47
     MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
48
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  1
                                                                                  2
    INTEGER, INTENT(IN) :: maxindex, maxedges
                                                                                  3
    INTEGER, INTENT(OUT) :: index(maxindex), edges(maxedges)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
                                                                                  5
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
                                                                                  6
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  7
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
                                                                                  8
    INTEGER, INTENT(OUT) :: newrank
                                                                                  9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
                                                                                  11
MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  12
                                                                                  13
    INTEGER, INTENT(IN) :: rank, maxneighbors
                                                                                  14
    INTEGER, INTENT(OUT) :: neighbors(maxneighbors)
                                                                                  15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  16
MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
                                                                                  17
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  18
    INTEGER, INTENT(IN) :: rank
                                                                                  19
    INTEGER, INTENT(OUT) :: nneighbors
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
                                                                                  22
MPI_Graphdims_get(comm, nnodes, nedges, ierror)
                                                                                  23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  24
    INTEGER, INTENT(OUT) :: nnodes, nedges
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  27
             recvtype, comm, request, ierror)
                                                                                  28
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  29
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  31
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  32
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
                                                                                  36
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  37
             recvtype, comm, request, ierror) !(_c)
                                                                                  38
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  39
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  41
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  43
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  45
MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
                                                                                  46
             displs, recvtype, comm, request, ierror)
                                                                                  47
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  48
```

1 INTEGER, INTENT(IN) :: sendcount 2 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 3 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: recvbuf 4 INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(\*), displs(\*) 5TYPE(MPI\_Comm), INTENT(IN) :: comm 6 TYPE(MPI\_Request), INTENT(OUT) :: request 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 MPI\_Ineighbor\_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, 9 displs, recvtype, comm, request, ierror) !(\_c) 10 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 11 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount 12TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 13 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 14 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(\*) 15INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN), ASYNCHRONOUS :: displs(\*) 16 TYPE(MPI\_Comm), INTENT(IN) :: comm 17 TYPE(MPI\_Request), INTENT(OUT) :: request 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 20MPI\_Ineighbor\_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 21recvtype, comm, request, ierror) 22 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 23INTEGER, INTENT(IN) :: sendcount, recvcount  $^{24}$ TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 25TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 26TYPE(MPI\_Comm), INTENT(IN) :: comm 27TYPE(MPI\_Request), INTENT(OUT) :: request 28INTEGER, OPTIONAL, INTENT(OUT) :: ierror 29 MPI\_Ineighbor\_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, 30 recvtype, comm, request, ierror) !(\_c) 31TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 32 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: sendcount, recvcount 33 TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 34 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 35 TYPE(MPI\_Comm), INTENT(IN) :: comm 36 TYPE(MPI\_Request), INTENT(OUT) :: request 37 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 38 39 MPI\_Ineighbor\_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, 40recvcounts, rdispls, recvtype, comm, request, ierror) 41 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf 42INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(\*), sdispls(\*), 43 recvcounts(\*), rdispls(\*) 44TYPE(MPI\_Datatype), INTENT(IN) :: sendtype, recvtype 45TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: recvbuf 46TYPE(MPI\_Comm), INTENT(IN) :: comm 47 TYPE(MPI\_Request), INTENT(OUT) :: request 48

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   1
                                                                                   2
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
              recvcounts, rdispls, recvtype, comm, request, ierror) !(_c)
                                                                                   4
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                   5
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                   6
              sendcounts(*), recvcounts(*)
                                                                                   7
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                   8
              rdispls(*)
                                                                                   9
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  10
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  12
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  14
                                                                                  15
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  16
              recvcounts, rdispls, recvtypes, comm, request, ierror)
                                                                                  17
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  18
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                  19
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  20
              rdispls(*)
                                                                                  21
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  22
              recvtypes(*)
                                                                                  23
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  24
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  28
              recvcounts, rdispls, recvtypes, comm, request, ierror) !(_c)
                                                                                  29
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  30
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                  31
              sendcounts(*), recvcounts(*)
                                                                                  32
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  33
              rdispls(*)
                                                                                  34
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  35
              recvtypes(*)
                                                                                  36
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  37
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  38
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  40
                                                                                  41
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  42
              recvtype, comm, ierror)
                                                                                  43
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  44
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  45
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  46
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  47
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
3
                   recvtype, comm, ierror) !(_c)
4
         TYPE(*), DIMENSION(...), INTENT(IN) :: sendbuf
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
6
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
7
         TYPE(*), DIMENSION(...) :: recvbuf
8
         TYPE(MPI_Comm), INTENT(IN) :: comm
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
12
                   recvcount, recvtype, comm, info, request, ierror)
13
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
14
         INTEGER, INTENT(IN) :: sendcount, recvcount
15
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
16
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
17
         TYPE(MPI_Comm), INTENT(IN) :: comm
18
         TYPE(MPI_Info), INTENT(IN) :: info
19
         TYPE(MPI_Request), INTENT(OUT) :: request
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
22
                   recvcount, recvtype, comm, info, request, ierror) !(_c)
23
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
24
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
25
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recytype
26
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(MPI_Info), INTENT(IN) :: info
29
         TYPE(MPI_Request), INTENT(OUT) :: request
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
32
     MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
33
                   displs, recvtype, comm, ierror)
34
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
35
         INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*)
36
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recytype
37
         TYPE(*), DIMENSION(...) :: recvbuf
38
         TYPE(MPI_Comm), INTENT(IN) :: comm
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
41
                   displs, recvtype, comm, ierror) !(_c)
42
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
43
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcounts(*)
44
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
45
         TYPE(*), DIMENSION(...) :: recvbuf
46
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
47
         TYPE(MPI_Comm), INTENT(IN) :: comm
48
```

```
1
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  2
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
              recvcounts, displs, recvtype, comm, info, request, ierror)
                                                                                  4
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  5
    INTEGER, INTENT(IN) :: sendcount, displs(*)
                                                                                  6
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  7
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  8
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  9
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  10
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  11
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  12
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  13
                                                                                  14
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
                                                                                  15
              recvcounts, displs, recvtype, comm, info, request, ierror)
                                                                                  16
              !(_c)
                                                                                  17
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  18
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount
                                                                                  19
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  20
                                                                                  21
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
                                                                                  22
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: displs(*)
                                                                                  23
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  24
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  25
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  28
              recvtype, comm, ierror)
                                                                                  29
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  30
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  31
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  32
    TYPE(*), DIMENSION(...) :: recvbuf
                                                                                  33
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
                                                                                  36
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
                                                                                  37
              recvtype, comm, ierror) !(_c)
                                                                                  38
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  39
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  41
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  42
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
                                                                                  45
              recvcount, recvtype, comm, info, request, ierror)
                                                                                  46
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  47
    INTEGER, INTENT(IN) :: sendcount, recvcount
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
2
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
3
         TYPE(MPI_Comm), INTENT(IN) :: comm
4
         TYPE(MPI_Info), INTENT(IN) :: info
5
         TYPE(MPI_Request), INTENT(OUT) :: request
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
     MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
8
                   recvcount, recvtype, comm, info, request, ierror) !(_c)
9
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
10
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcount, recvcount
11
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
12
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
13
         TYPE(MPI_Comm), INTENT(IN) :: comm
14
         TYPE(MPI_Info), INTENT(IN) :: info
15
         TYPE(MPI_Request), INTENT(OUT) :: request
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
18
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
19
                   recvcounts, rdispls, recvtype, comm, ierror)
20
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
21
         INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
22
                   rdispls(*)
23
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
24
         TYPE(*), DIMENSION(...) :: recvbuf
25
         TYPE(MPI_Comm), INTENT(IN) :: comm
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
28
                   recvcounts, rdispls, recvtype, comm, ierror) !(_c)
29
         TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
30
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
31
                   recvcounts(*)
32
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
33
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
34
         TYPE(*), DIMENSION(..) :: recvbuf
35
         TYPE(MPI_Comm), INTENT(IN) :: comm
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
39
                   recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
40
                   ierror)
41
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
42
         INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
43
                   recvcounts(*), rdispls(*)
44
         TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
45
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
46
         TYPE(MPI_Comm), INTENT(IN) :: comm
47
         TYPE(MPI_Info), INTENT(IN) :: info
48
```

```
1
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  2
                                                                                  3
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
                                                                                  4
              recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
                                                                                  5
              ierror) !(_c)
                                                                                  6
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  7
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
                                                                                  8
              sendcounts(*), recvcounts(*)
                                                                                  9
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  10
              rdispls(*)
                                                                                  11
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
                                                                                  12
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  13
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  14
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  15
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  17
                                                                                  18
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  19
              recvcounts, rdispls, recvtypes, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  20
    INTEGER, INTENT(IN) :: sendcounts(*), recvcounts(*)
                                                                                  21
                                                                                  22
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                  23
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                  24
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  25
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  27
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
                                                                                  28
              recvcounts, rdispls, recvtypes, comm, ierror) !(_c)
                                                                                  29
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
                                                                                  30
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: sendcounts(*),
                                                                                  31
              recvcounts(*)
                                                                                  32
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: sdispls(*), rdispls(*)
                                                                                  33
    TYPE(MPI_Datatype), INTENT(IN) :: sendtypes(*), recvtypes(*)
                                                                                  34
    TYPE(*), DIMENSION(..) :: recvbuf
                                                                                  35
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  37
                                                                                  38
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
                                                                                  39
              recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
                                                                                  40
              ierror)
                                                                                  41
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
                                                                                  42
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
                                                                                  43
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
                                                                                  44
              rdispls(*)
                                                                                  45
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
                                                                                  46
              recvtypes(*)
                                                                                  47
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
2
         TYPE(MPI_Info), INTENT(IN) :: info
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
6
                   recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
7
                   ierror) !(_c)
8
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: sendbuf
9
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN), ASYNCHRONOUS ::
10
                    sendcounts(*), recvcounts(*)
11
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
12
                    rdispls(*)
13
         TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
14
                    recvtypes(*)
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: recvbuf
16
         TYPE(MPI_Comm), INTENT(IN) :: comm
17
         TYPE(MPI_Info), INTENT(IN) :: info
18
         TYPE(MPI_Request), INTENT(OUT) :: request
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_Topo_test(comm, status, ierror)
22
         TYPE(MPI_Comm), INTENT(IN) :: comm
23
         INTEGER, INTENT(OUT) :: status
^{24}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
26
     A.4.7 MPI Environmental Management Fortran 2008 Bindings
27
28
     DOUBLE PRECISION MPI_Wtick()
29
    DOUBLE PRECISION MPI_Wtime()
30
^{31}
    MPI_Add_error_class(errorclass, ierror)
32
         INTEGER, INTENT(OUT) :: errorclass
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Add_error_code(errorclass, errorcode, ierror)
35
         INTEGER, INTENT(IN) :: errorclass
36
         INTEGER, INTENT(OUT) :: errorcode
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Add_error_string(errorcode, string, ierror)
40
         INTEGER, INTENT(IN) :: errorcode
41
         CHARACTER(LEN=*), INTENT(IN) :: string
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
     MPI_Alloc_mem(size, info, baseptr, ierror)
44
45
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
46
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
47
         TYPE(MPI_Info), INTENT(IN) :: info
48
         TYPE(C_PTR), INTENT(OUT) :: baseptr
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
<pre>MPI_Comm_call_errhandler(comm, errorcode, ierror)</pre>	2
TYPE(MPI_Comm), INTENT(IN) :: comm	3
INTEGER, INTENT(IN) :: errorcode	4
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	5
MDT Commence on an annual on (commence on the annual on isomer)	7
<pre>MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)</pre>	8
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	9
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	10
	11
MPI_Comm_get_errhandler(comm, errhandler, ierror)	12
TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	13
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	14
	15 16
MPI_Comm_set_errhandler(comm, errhandler, ierror)	10
TYPE(MPI_Comm), INTENT(IN) :: comm TYPE(MPI_Errhandler), INTENT(IN) :: errhandler	18
INTEGER, OPTIONAL, INTENT(OUT) :: errhandler	19
	20
MPI_Errhandler_free(errhandler, ierror)	21
TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler	22
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	23
<pre>MPI_Error_class(errorcode, errorclass, ierror)</pre>	24
INTEGER, INTENT(IN) :: errorcode	25 26
INTEGER, INTENT(OUT) :: errorclass	20
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	28
MPI_Error_string(errorcode, string, resultlen, ierror)	29
INTEGER, INTENT(IN) :: errorcode	30
CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string	31
INTEGER, INTENT(OUT) :: resultlen	32
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	33
<pre>MPI_File_call_errhandler(fh, errorcode, ierror)</pre>	34
TYPE(MPI_File), INTENT(IN) :: fh	35 36
INTEGER, INTENT(IN) :: errorcode	37
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	38
<pre>MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)</pre>	39
<pre>PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn</pre>	40
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	41
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	42
MPI_File_get_errhandler(file, errhandler, ierror)	43
TYPE(MPI_File), INTENT(IN) :: file	44
TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler	45 46
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
MPI_File_set_errhandler(file, errhandler, ierror)	48

```
1
         TYPE(MPI_File), INTENT(IN) :: file
\mathbf{2}
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Free_mem(base, ierror)
5
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: base
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_Get_library_version(version, resultlen, ierror)
9
         CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
10
         INTEGER, INTENT(OUT) :: resultlen
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Get_processor_name(name, resultlen, ierror)
13
         CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
14
         INTEGER, INTENT(OUT) :: resultlen
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
17
     MPI_Get_version(version, subversion, ierror)
18
         INTEGER, INTENT(OUT) :: version, subversion
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
    MPI_Session_call_errhandler(session, errorcode, ierror)
21
         TYPE(MPI_Session), INTENT(IN) :: session
22
         INTEGER, INTENT(IN) :: errorcode
23
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{24}
25
    MPI_Session_create_errhandler(session_errhandler_fn, errhandler, ierror)
26
         PROCEDURE(MPI_Session_errhandler_function) :: session_errhandler_fn
27
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
    MPI_Session_get_errhandler(session, errhandler, ierror)
30
         TYPE(MPI_Session), INTENT(IN) :: session
31
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
34
     MPI_Session_set_errhandler(session, errhandler, ierror)
35
         TYPE(MPI_Session), INTENT(IN) :: session
36
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_Win_call_errhandler(win, errorcode, ierror)
39
         TYPE(MPI_Win), INTENT(IN) :: win
40
         INTEGER, INTENT(IN) :: errorcode
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
43
     MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
44
         PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
45
         TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
46
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
47
    MPI_Win_get_errhandler(win, errhandler, ierror)
48
```

```
TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   1
                                                                                   2
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
MPI_Win_set_errhandler(win, errhandler, ierror)
                                                                                   5
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                   6
    TYPE(MPI Errhandler), INTENT(IN) :: errhandler
                                                                                   7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   8
                                                                                   9
                                                                                   10
A.4.8 The Info Object Fortran 2008 Bindings
                                                                                   11
MPI_Info_create(info, ierror)
                                                                                   12
    TYPE(MPI_Info), INTENT(OUT) :: info
                                                                                   13
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   14
                                                                                   15
MPI_Info_create_env(info, ierror)
                                                                                   16
    TYPE(MPI_Info), INTENT(OUT) :: info
                                                                                   17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   18
MPI_Info_delete(info, key, ierror)
                                                                                   19
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   20
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                   21
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   22
                                                                                   23
MPI_Info_dup(info, newinfo, ierror)
                                                                                   24
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   25
    TYPE(MPI_Info), INTENT(OUT) :: newinfo
                                                                                   26
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   27
MPI_Info_free(info, ierror)
                                                                                   28
    TYPE(MPI_Info), INTENT(INOUT) :: info
                                                                                   29
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   30
                                                                                   31
MPI_Info_get_nkeys(info, nkeys, ierror)
                                                                                   32
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   33
    INTEGER, INTENT(OUT) :: nkeys
                                                                                   34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   35
MPI_Info_get_nthkey(info, n, key, ierror)
                                                                                   36
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   37
    INTEGER, INTENT(IN) :: n
                                                                                   38
    CHARACTER(LEN=*), INTENT(OUT) :: key
                                                                                   39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   40
                                                                                   41
MPI_Info_get_string(info, key, buflen, value, flag, ierror)
                                                                                   42
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                   43
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                   44
    INTEGER, INTENT(INOUT) :: buflen
                                                                                   45
    CHARACTER(LEN=*), INTENT(OUT) :: value
                                                                                   46
    LOGICAL, INTENT(OUT) :: flag
                                                                                   47
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   48
```

```
1
     MPI_Info_set(info, key, value, ierror)
\mathbf{2}
         TYPE(MPI_Info), INTENT(IN) :: info
3
         CHARACTER(LEN=*), INTENT(IN) :: key, value
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
6
     A.4.9 Process Creation and Management Fortran 2008 Bindings
7
8
    MPI_Abort(comm, errorcode, ierror)
9
         TYPE(MPI_Comm), INTENT(IN) :: comm
10
         INTEGER, INTENT(IN) :: errorcode
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_Close_port(port_name, ierror)
13
         CHARACTER(LEN=*), INTENT(IN) :: port_name
14
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
15
16
     MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
17
         CHARACTER(LEN=*), INTENT(IN) :: port_name
18
         TYPE(MPI_Info), INTENT(IN) :: info
19
         INTEGER, INTENT(IN) :: root
20
         TYPE(MPI_Comm), INTENT(IN) :: comm
21
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
22
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
23
     MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
^{24}
         CHARACTER(LEN=*), INTENT(IN) :: port_name
25
         TYPE(MPI_Info), INTENT(IN) :: info
26
         INTEGER, INTENT(IN) :: root
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(MPI_Comm), INTENT(OUT) :: newcomm
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_Comm_disconnect(comm, ierror)
32
         TYPE(MPI_Comm), INTENT(INOUT) :: comm
33
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
34
     MPI_Comm_get_parent(parent, ierror)
35
         TYPE(MPI_Comm), INTENT(OUT) :: parent
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
38
     MPI_Comm_join(fd, intercomm, ierror)
39
         INTEGER, INTENT(IN) :: fd
40
         TYPE(MPI_Comm), INTENT(OUT) :: intercomm
41
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
42
     MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
43
44
                   array_of_errcodes, ierror)
45
         CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
         INTEGER, INTENT(IN) :: maxprocs, root
46
47
         TYPE(MPI_Info), INTENT(IN) :: info
48
         TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
1
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
                                                                                   2
    INTEGER :: array_of_errcodes(*)
                                                                                   3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   4
MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
                                                                                   5
              array_of_maxprocs, array_of_info, root, comm, intercomm,
                                                                                   6
              array_of_errcodes, ierror)
                                                                                   7
    INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
                                                                                   8
    CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
                                                                                   9
              array_of_argv(count, *)
                                                                                   10
    TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
                                                                                   11
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                   12
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
                                                                                   13
    INTEGER :: array_of_errcodes(*)
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   15
                                                                                   16
MPI_Finalize(ierror)
                                                                                   17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   18
MPI_Finalized(flag, ierror)
                                                                                   19
    LOGICAL, INTENT(OUT) :: flag
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
                                                                                  22
MPI_Init(ierror)
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
MPI_Init_thread(required, provided, ierror)
                                                                                  25
    INTEGER, INTENT(IN) :: required
                                                                                   26
    INTEGER, INTENT(OUT) :: provided
                                                                                  27
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  28
                                                                                  29
MPI_Initialized(flag, ierror)
                                                                                  30
    LOGICAL, INTENT(OUT) :: flag
                                                                                   31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   32
MPI_Is_thread_main(flag, ierror)
                                                                                   33
    LOGICAL, INTENT(OUT) :: flag
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
                                                                                  36
MPI_Lookup_name(service_name, info, port_name, ierror)
                                                                                  37
    CHARACTER(LEN=*), INTENT(IN) :: service_name
                                                                                  38
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  39
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                   40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
MPI_Open_port(info, port_name, ierror)
                                                                                  42
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  43
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
                                                                                  44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  45
                                                                                  46
MPI_Publish_name(service_name, info, port_name, ierror)
                                                                                   47
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
                                                                                  48
```

```
1
         TYPE(MPI_Info), INTENT(IN) :: info
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_Query_thread(provided, ierror)
4
         INTEGER, INTENT(OUT) :: provided
5
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
\overline{7}
     MPI_Session_finalize(session, ierror)
8
         TYPE(MPI_Session), INTENT(INOUT) :: session
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
     MPI_Session_get_info(session, info_used, ierror)
11
         TYPE(MPI_Session), INTENT(IN) :: session
12
         TYPE(MPI_Info), INTENT(OUT) :: info_used
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
     MPI_Session_get_nth_pset(session, info, n, pset_len, pset_name, ierror)
16
         TYPE(MPI_Session), INTENT(IN) :: session
17
         TYPE(MPI_Info), INTENT(IN) :: info
18
         INTEGER, INTENT(IN) :: n
19
         INTEGER, INTENT(INOUT) :: pset_len
20
         CHARACTER(LEN=*), INTENT(OUT) :: pset_name
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_Session_get_num_psets(session, info, npset_names, ierror)
23
         TYPE(MPI_Session), INTENT(IN) :: session
24
         TYPE(MPI_Info), INTENT(IN) :: info
25
         INTEGER, INTENT(OUT) :: npset_names
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
28
     MPI_Session_get_pset_info(session, pset_name, info, ierror)
29
         TYPE(MPI_Session), INTENT(IN) :: session
30
         CHARACTER(LEN=*), INTENT(IN) :: pset_name
31
         TYPE(MPI_Info), INTENT(OUT) :: info
32
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
33
     MPI_Session_init(info, errhandler, session, ierror)
34
         TYPE(MPI_Info), INTENT(IN) :: info
35
         TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
36
         TYPE(MPI_Session), INTENT(OUT) :: session
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
     MPI_Unpublish_name(service_name, info, port_name, ierror)
40
         CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
41
         TYPE(MPI_Info), INTENT(IN) :: info
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
44
     A.4.10 One-Sided Communications Fortran 2008 Bindings
45
46
     MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
47
                   target_disp, target_count, target_datatype, op, win, ierror)
48
```

TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 1  $\mathbf{2}$ INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 3 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 4 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp TYPE(MPI\_Op), INTENT(IN) :: op 5TYPE(MPI\_Win), INTENT(IN) :: win 6 7 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 8 MPI\_Accumulate(origin\_addr, origin\_count, origin\_datatype, target\_rank, 9 target\_disp, target\_count, target\_datatype, op, win, ierror) 10 !(\_c) 11 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 12INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: origin\_count, target\_count 13 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 14 INTEGER, INTENT(IN) :: target\_rank 15INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 16 TYPE(MPI\_Op), INTENT(IN) :: op 17TYPE(MPI\_Win), INTENT(IN) :: win 18 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 19 20MPI\_Compare\_and\_swap(origin\_addr, compare\_addr, result\_addr, datatype, 21target\_rank, target\_disp, win, ierror) 22 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr, 23compare\_addr 24TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: result\_addr 25TYPE(MPI\_Datatype), INTENT(IN) :: datatype 26INTEGER, INTENT(IN) :: target\_rank 27INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 28 TYPE(MPI\_Win), INTENT(IN) :: win 29 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 30 MPI\_Fetch\_and\_op(origin\_addr, result\_addr, datatype, target\_rank, 31target\_disp, op, win, ierror) 32 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 33 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: result\_addr 34 TYPE(MPI\_Datatype), INTENT(IN) :: datatype 35 INTEGER, INTENT(IN) :: target\_rank 36 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 37 TYPE(MPI\_Op), INTENT(IN) :: op 38 TYPE(MPI\_Win), INTENT(IN) :: win 39 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 4041 MPI\_Get(origin\_addr, origin\_count, origin\_datatype, target\_rank, 42target\_disp, target\_count, target\_datatype, win, ierror) 43 TYPE(\*), DIMENSION(..), ASYNCHRONOUS :: origin\_addr 44INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 45TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 46INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 47TYPE(MPI\_Win), INTENT(IN) :: win 48

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
2
     MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
3
                   target_disp, target_count, target_datatype, win, ierror) !(_c)
4
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
5
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
6
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
7
         INTEGER, INTENT(IN) :: target_rank
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
9
         TYPE(MPI_Win), INTENT(IN) :: win
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
13
                  result_count, result_datatype, target_rank, target_disp,
14
                  target_count, target_datatype, op, win, ierror)
15
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
16
         INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
17
                   target_count
18
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
19
                   target_datatype
20
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
21
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
22
         TYPE(MPI_Op), INTENT(IN) :: op
23
         TYPE(MPI_Win), INTENT(IN) :: win
^{24}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
     MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
26
                  result_count, result_datatype, target_rank, target_disp,
27
                  target_count, target_datatype, op, win, ierror) !(_c)
28
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin_addr
29
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, result_count,
30
                   target_count
31
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, result_datatype,
32
                   target_datatype
33
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: result_addr
34
         INTEGER, INTENT(IN) :: target_rank
35
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
36
         TYPE(MPI_Op), INTENT(IN) :: op
37
         TYPE(MPI_Win), INTENT(IN) :: win
38
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
39
40
     MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
41
                  target_disp, target_count, target_datatype, win, ierror)
42
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
43
         INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
44
         TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
45
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
46
         TYPE(MPI_Win), INTENT(IN) :: win
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  2
              target_disp, target_count, target_datatype, win, ierror) !(_c)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  4
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  5
                                                                                  6
    INTEGER, INTENT(IN) :: target_rank
                                                                                  7
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  9
                                                                                  10
MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  11
             target_disp, target_count, target_datatype, op, win, request,
                                                                                  12
              ierror)
                                                                                  13
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  14
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  15
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  16
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  17
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  18
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  19
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
                                                                                  22
MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  23
             target_disp, target_count, target_datatype, op, win, request,
                                                                                  24
             ierror) !(_c)
                                                                                  25
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  26
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
                                                                                  27
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  28
    INTEGER, INTENT(IN) :: target_rank
                                                                                  29
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  30
    TYPE(MPI_Op), INTENT(IN) :: op
                                                                                  31
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  32
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  33
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  34
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  35
             target_disp, target_count, target_datatype, win, request,
                                                                                  36
             ierror)
                                                                                  37
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: origin_addr
                                                                                  38
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
                                                                                  39
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  40
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  41
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  42
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  44
                                                                                  45
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  46
             target_disp, target_count, target_datatype, win, request,
                                                                                  47
              ierror) !( c)
                                                                                  48
```

1 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: origin\_addr 2 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: origin\_count, target\_count 3 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 4 INTEGER, INTENT(IN) :: target\_rank 5INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 6 TYPE(MPI\_Win), INTENT(IN) :: win 7 TYPE(MPI\_Request), INTENT(OUT) :: request 8 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 9 MPI\_Rget\_accumulate(origin\_addr, origin\_count, origin\_datatype, 10 result\_addr, result\_count, result\_datatype, target\_rank, 11 target\_disp, target\_count, target\_datatype, op, win, request, 12ierror) 13 TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 14 INTEGER, INTENT(IN) :: origin\_count, result\_count, target\_rank, 15target\_count 16TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, result\_datatype, 17 target\_datatype 18 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: result\_addr 19 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 20TYPE(MPI\_Op), INTENT(IN) :: op 21TYPE(MPI\_Win), INTENT(IN) :: win 22 TYPE(MPI\_Request), INTENT(OUT) :: request 23 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 2425MPI\_Rget\_accumulate(origin\_addr, origin\_count, origin\_datatype, 26result\_addr, result\_count, result\_datatype, target\_rank, 27target\_disp, target\_count, target\_datatype, op, win, request, 28ierror) !( c) 29TYPE(\*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin\_addr 30 INTEGER(KIND=MPI\_COUNT\_KIND), INTENT(IN) :: origin\_count, result\_count, 31target\_count 32 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, result\_datatype, 33 target\_datatype 34 TYPE(\*), DIMENSION(...), ASYNCHRONOUS :: result\_addr 35INTEGER, INTENT(IN) :: target\_rank 36 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 37 TYPE(MPI\_Op), INTENT(IN) :: op 38 TYPE(MPI\_Win), INTENT(IN) :: win 39 TYPE(MPI\_Request), INTENT(OUT) :: request 40 INTEGER, OPTIONAL, INTENT(OUT) :: ierror 41 MPI\_Rput(origin\_addr, origin\_count, origin\_datatype, target\_rank, 42target\_disp, target\_count, target\_datatype, win, request, 43 ierror) 44 TYPE(\*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: origin\_addr 45 INTEGER, INTENT(IN) :: origin\_count, target\_rank, target\_count 46 TYPE(MPI\_Datatype), INTENT(IN) :: origin\_datatype, target\_datatype 47 INTEGER(KIND=MPI\_ADDRESS\_KIND), INTENT(IN) :: target\_disp 48

```
TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  1
                                                                                  2
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  3
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
                                                                                  5
             target_disp, target_count, target_datatype, win, request,
                                                                                  6
             ierror) !( c)
                                                                                  7
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
                                                                                  8
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: origin_count, target_count
                                                                                  0
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
                                                                                  10
    INTEGER, INTENT(IN) :: target_rank
                                                                                  11
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
                                                                                  12
    TYPE(MPI_Win), INTENT(IN) :: win
                                                                                  13
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
                                                                                  16
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
                                                                                  17
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                  18
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
                                                                                  19
    INTEGER, INTENT(IN) :: disp_unit
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  20
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  21
                                                                                  22
    TYPE(C_PTR), INTENT(OUT) :: baseptr
                                                                                  23
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror) !(_c)
                                                                                  26
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                  27
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
                                                                                  28
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  29
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  30
    TYPE(C_PTR), INTENT(OUT) :: baseptr
                                                                                  31
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
                                                                                  34
MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
                                                                                  35
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                  36
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
                                                                                  37
    INTEGER, INTENT(IN) :: disp_unit
                                                                                  38
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  39
    TYPE(MPI_Comm), INTENT(IN) :: comm
                                                                                  40
    TYPE(C_PTR), INTENT(OUT) :: baseptr
                                                                                  41
    TYPE(MPI_Win), INTENT(OUT) :: win
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
                                                                                  44
              !(_c)
                                                                                  45
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                  46
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
                                                                                  47
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                  48
```

```
1
         TYPE(MPI_Comm), INTENT(IN) :: comm
\mathbf{2}
         TYPE(C_PTR), INTENT(OUT) :: baseptr
3
         TYPE(MPI_Win), INTENT(OUT) :: win
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_Win_attach(win, base, size, ierror)
6
         TYPE(MPI_Win), INTENT(IN) :: win
7
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
8
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_Win_complete(win, ierror)
12
         TYPE(MPI_Win), INTENT(IN) :: win
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
    MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
15
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
16
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
17
         INTEGER, INTENT(IN) :: disp_unit
18
         TYPE(MPI_Info), INTENT(IN) :: info
19
         TYPE(MPI_Comm), INTENT(IN) :: comm
20
         TYPE(MPI_Win), INTENT(OUT) :: win
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Win_create(base, size, disp_unit, info, comm, win, ierror) !(_c)
^{24}
         TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
25
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size, disp_unit
26
         TYPE(MPI_Info), INTENT(IN) :: info
27
         TYPE(MPI_Comm), INTENT(IN) :: comm
28
         TYPE(MPI_Win), INTENT(OUT) :: win
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
     MPI_Win_create_dynamic(info, comm, win, ierror)
^{31}
         TYPE(MPI_Info), INTENT(IN) :: info
32
         TYPE(MPI_Comm), INTENT(IN) :: comm
33
         TYPE(MPI_Win), INTENT(OUT) :: win
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_Win_detach(win, base, ierror)
37
         TYPE(MPI_Win), INTENT(IN) :: win
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: base
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Win_fence(assert, win, ierror)
41
         INTEGER, INTENT(IN) :: assert
42
         TYPE(MPI_Win), INTENT(IN) :: win
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Win_flush(rank, win, ierror)
46
         INTEGER, INTENT(IN) :: rank
47
         TYPE(MPI_Win), INTENT(IN) :: win
48
```

INTEGER, OPTIONAL, INTENT(OUT) :: ierror	1
MPI_Win_flush_all(win, ierror)	2
TYPE(MPI_Win), INTENT(IN) :: win	3
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	4 5
	5
MPI_Win_flush_local(rank, win, ierror)	7
INTEGER, INTENT(IN) :: rank	8
TYPE(MPI_Win), INTENT(IN) :: win INTEGER, OPTIONAL, INTENT(OUT) :: ierror	9
INTEGER, OFITONAL, INTENI(001) TETTOT	10
<pre>MPI_Win_flush_local_all(win, ierror)</pre>	11
TYPE(MPI_Win), INTENT(IN) :: win	12
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	13
MPI_Win_free(win, ierror)	14
TYPE(MPI_Win), INTENT(INOUT) :: win	15
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	16
	17
MPI_Win_get_group(win, group, ierror)	18
TYPE(MPI_Win), INTENT(IN) :: win	19
TYPE(MPI_Group), INTENT(OUT) :: group INTEGER, OPTIONAL, INTENT(OUT) :: ierror	20
INTEGER, OFITONAL, INTENT(001) TETTOT	21
<pre>MPI_Win_get_info(win, info_used, ierror)</pre>	22
TYPE(MPI_Win), INTENT(IN) :: win	23 24
TYPE(MPI_Info), INTENT(OUT) :: info_used	24 25
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	25
MPI_Win_lock(lock_type, rank, assert, win, ierror)	20
INTEGER, INTENT(IN) :: lock_type, rank, assert	28
TYPE(MPI_Win), INTENT(IN) :: win	29
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	30
MDT Uin look all (accept win inverse)	31
<pre>MPI_Win_lock_all(assert, win, ierror)</pre>	32
TYPE(MPI_Win), INTENT(IN) :: win	33
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	34
	35
<pre>MPI_Win_post(group, assert, win, ierror)</pre>	36
TYPE(MPI_Group), INTENT(IN) :: group	37
INTEGER, INTENT(IN) :: assert	38
TYPE(MPI_Win), INTENT(IN) :: win	39
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	40
MPI_Win_set_info(win, info, ierror)	41
TYPE(MPI_Win), INTENT(IN) :: win	42
TYPE(MPI_Info), INTENT(IN) :: info	43 44
INTEGER, OPTIONAL, INTENT(OUT) :: ierror	44 45
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)	43 46
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR	47
TYPE(MPI_Win), INTENT(IN) :: win	48

```
1
         INTEGER, INTENT(IN) :: rank
2
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
3
         INTEGER, INTENT(OUT) :: disp_unit
4
         TYPE(C_PTR), INTENT(OUT) :: baseptr
\mathbf{5}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
6
     MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror) !(_c)
7
         USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
8
         TYPE(MPI_Win), INTENT(IN) :: win
9
         INTEGER, INTENT(IN) :: rank
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size, disp_unit
11
         TYPE(C_PTR), INTENT(OUT) :: baseptr
12
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
13
14
     MPI_Win_start(group, assert, win, ierror)
15
         TYPE(MPI_Group), INTENT(IN) :: group
16
         INTEGER, INTENT(IN) :: assert
17
         TYPE(MPI_Win), INTENT(IN) :: win
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     MPI_Win_sync(win, ierror)
20
         TYPE(MPI_Win), INTENT(IN) :: win
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
23
     MPI_Win_test(win, flag, ierror)
^{24}
         TYPE(MPI_Win), INTENT(IN) :: win
25
         LOGICAL, INTENT(OUT) :: flag
26
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
27
     MPI_Win_unlock(rank, win, ierror)
28
         INTEGER, INTENT(IN) :: rank
29
         TYPE(MPI_Win), INTENT(IN) :: win
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
^{31}
32
     MPI_Win_unlock_all(win, ierror)
33
         TYPE(MPI_Win), INTENT(IN) :: win
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Win_wait(win, ierror)
36
         TYPE(MPI_Win), INTENT(IN) :: win
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
39
40
     A.4.11 External Interfaces Fortran 2008 Bindings
41
42
     MPI_Grequest_complete(request, ierror)
         TYPE(MPI_Request), INTENT(IN) :: request
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
46
                   ierror)
47
         PROCEDURE(MPI_Grequest_query_function) :: query_fn
48
```

```
1
    PROCEDURE(MPI_Grequest_free_function) :: free_fn
                                                                                   2
    PROCEDURE(MPI_Grequest_cancel_function) :: cancel_fn
                                                                                   3
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
                                                                                   4
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   5
                                                                                   6
MPI_Status_set_cancelled(status, flag, ierror)
                                                                                   7
    TYPE(MPI_Status), INTENT(INOUT) :: status
                                                                                   8
    LOGICAL, INTENT(IN) :: flag
                                                                                   9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   10
                                                                                   11
MPI_Status_set_elements(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(INOUT) :: status
                                                                                  12
                                                                                  13
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  14
    INTEGER, INTENT(IN) :: count
                                                                                   15
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   16
MPI_Status_set_elements_x(status, datatype, count, ierror)
                                                                                   17
    TYPE(MPI_Status), INTENT(INOUT) :: status
                                                                                  18
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  19
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  20
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  21
                                                                                  22
                                                                                  23
A.4.12 I/O Fortran 2008 Bindings
                                                                                  24
MPI_CONVERSION_FN_NULL (userbuf, datatype, count, filebuf, position,
                                                                                  25
              extra_state, ierror)
                                                                                  26
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                  27
    TYPE(C_PTR), VALUE :: userbuf, filebuf
                                                                                  28
    TYPE(MPI_Datatype) :: datatype
                                                                                  29
    INTEGER :: count, ierror
                                                                                  30
    INTEGER(KIND=MPI_OFFSET_KIND) :: position
                                                                                  31
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                  32
                                                                                  33
MPI_CONVERSION_FN_NULL_C(userbuf, datatype, count, filebuf, position,
                                                                                  34
              extra_state, ierror) !(_c)
                                                                                  35
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
                                                                                  36
    TYPE(C_PTR), VALUE :: userbuf, filebuf
                                                                                  37
    TYPE(MPI_Datatype) :: datatype
                                                                                  38
    INTEGER(KIND=MPI_COUNT_KIND) :: count
                                                                                   39
    INTEGER(KIND=MPI_OFFSET_KIND) :: position
                                                                                   40
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
                                                                                   41
    INTEGER :: ierror
                                                                                  42
MPI_File_close(fh, ierror)
                                                                                   43
    TYPE(MPI_File), INTENT(INOUT) :: fh
                                                                                   44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   45
                                                                                   46
MPI_File_delete(filename, info, ierror)
                                                                                   47
    CHARACTER(LEN=*), INTENT(IN) :: filename
                                                                                   48
```

```
1
         TYPE(MPI_Info), INTENT(IN) :: info
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_File_get_amode(fh, amode, ierror)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         INTEGER, INTENT(OUT) :: amode
6
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
7
8
     MPI_File_get_atomicity(fh, flag, ierror)
9
         TYPE(MPI_File), INTENT(IN) :: fh
10
         LOGICAL, INTENT(OUT) :: flag
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
     MPI_File_get_byte_offset(fh, offset, disp, ierror)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
15
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
16
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
17
^{18}
     MPI_File_get_group(fh, group, ierror)
19
         TYPE(MPI_File), INTENT(IN) :: fh
20
         TYPE(MPI_Group), INTENT(OUT) :: group
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_File_get_info(fh, info_used, ierror)
23
         TYPE(MPI_File), INTENT(IN) :: fh
24
         TYPE(MPI_Info), INTENT(OUT) :: info_used
25
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
26
27
     MPI_File_get_position(fh, offset, ierror)
28
         TYPE(MPI_File), INTENT(IN) :: fh
29
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
30
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
31
    MPI_File_get_position_shared(fh, offset, ierror)
32
         TYPE(MPI_File), INTENT(IN) :: fh
33
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
36
     MPI_File_get_size(fh, size, ierror)
37
         TYPE(MPI_File), INTENT(IN) :: fh
38
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_File_get_type_extent(fh, datatype, extent, ierror)
41
         TYPE(MPI_File), INTENT(IN) :: fh
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
    MPI_File_get_type_extent(fh, datatype, extent, ierror) !(_c)
47
         TYPE(MPI_File), INTENT(IN) :: fh
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  2
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: extent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  3
                                                                                  4
MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
                                                                                  5
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  6
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
                                                                                  7
    TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
                                                                                  8
    CHARACTER(LEN=*), INTENT(OUT) :: datarep
                                                                                  9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  10
                                                                                  11
MPI_File_iread(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  12
                                                                                  13
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  14
    INTEGER, INTENT(IN) :: count
                                                                                  15
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  16
                                                                                  17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  18
MPI_File_iread(fh, buf, count, datatype, request, ierror) !(_c)
                                                                                  19
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  20
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  21
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  23
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
                                                                                  26
MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
                                                                                  27
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  28
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  29
    INTEGER, INTENT(IN) :: count
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  31
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  32
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  33
MPI_File_iread_all(fh, buf, count, datatype, request, ierror) !(_c)
                                                                                  34
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  35
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  36
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  37
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  38
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  39
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  40
                                                                                  41
MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                  42
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  43
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  44
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  45
    INTEGER, INTENT(IN) :: count
                                                                                  46
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  47
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  48
```

```
1
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
\mathbf{2}
     MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
3
         TYPE(MPI_File), INTENT(IN) :: fh
4
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
5
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
6
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
7
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
8
         TYPE(MPI_Request), INTENT(OUT) :: request
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
11
     MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
14
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
15
         INTEGER, INTENT(IN) :: count
16
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
17
         TYPE(MPI_Request), INTENT(OUT) :: request
18
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
19
     MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
20
                   !(_c)
21
         TYPE(MPI_File), INTENT(IN) :: fh
22
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
23
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
24
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
25
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
26
         TYPE(MPI_Request), INTENT(OUT) :: request
27
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
28
^{29}
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
30
         TYPE(MPI_File), INTENT(IN) :: fh
^{31}
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
32
         INTEGER, INTENT(IN) :: count
33
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
34
         TYPE(MPI_Request), INTENT(OUT) :: request
35
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
36
     MPI_File_iread_shared(fh, buf, count, datatype, request, ierror) !(_c)
37
         TYPE(MPI_File), INTENT(IN) :: fh
38
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
39
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
40
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
41
         TYPE(MPI_Request), INTENT(OUT) :: request
42
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
43
^{44}
     MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
45
         TYPE(MPI_File), INTENT(IN) :: fh
46
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
47
         INTEGER, INTENT(IN) :: count
48
```

```
1
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  2
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  4
MPI_File_iwrite(fh, buf, count, datatype, request, ierror) !(_c)
                                                                                  5
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  6
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  7
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  9
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 11
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
                                                                                  12
                                                                                  13
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  14
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  15
    INTEGER, INTENT(IN) :: count
                                                                                  16
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  17
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  19
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror) !(_c)
                                                                                  20
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 21
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  22
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  24
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  26
                                                                                 27
MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
                                                                                 28
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  29
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                 30
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  31
    INTEGER, INTENT(IN) :: count
                                                                                  32
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  33
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  34
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  35
MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror) !(_c)
                                                                                  36
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  37
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  38
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  39
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  40
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  41
    TYPE(MPI_Request), INTENT(OUT) :: request
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
                                                                                 44
MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
                                                                                  45
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  46
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  47
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  48
```

```
1
         INTEGER, INTENT(IN) :: count
2
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Request), INTENT(OUT) :: request
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
6
                   !( c)
7
         TYPE(MPI_File), INTENT(IN) :: fh
8
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
9
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
10
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
11
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
12
         TYPE(MPI_Request), INTENT(OUT) :: request
13
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
14
15
    MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
16
         TYPE(MPI_File), INTENT(IN) :: fh
17
         TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
18
         INTEGER, INTENT(IN) :: count
19
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
20
         TYPE(MPI_Request), INTENT(OUT) :: request
21
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
22
     MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror) !(_c)
23
         TYPE(MPI_File), INTENT(IN) :: fh
24
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
25
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Request), INTENT(OUT) :: request
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_File_open(comm, filename, amode, info, fh, ierror)
^{31}
         TYPE(MPI_Comm), INTENT(IN) :: comm
32
         CHARACTER(LEN=*), INTENT(IN) :: filename
33
         INTEGER, INTENT(IN) :: amode
34
         TYPE(MPI_Info), INTENT(IN) :: info
35
         TYPE(MPI_File), INTENT(OUT) :: fh
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
    MPI_File_preallocate(fh, size, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
40
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
41
42
     MPI_File_read(fh, buf, count, datatype, status, ierror)
43
         TYPE(MPI_File), INTENT(IN) :: fh
44
         TYPE(*), DIMENSION(..) :: buf
45
         INTEGER, INTENT(IN) :: count
46
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
47
         TYPE(MPI_Status) :: status
48
```

```
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                   1
                                                                                  2
MPI_File_read(fh, buf, count, datatype, status, ierror) !(_c)
                                                                                   3
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  4
    TYPE(*), DIMENSION(..) :: buf
                                                                                  5
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  6
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  7
    TYPE(MPI_Status) :: status
                                                                                   8
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  9
                                                                                  10
MPI_File_read_all(fh, buf, count, datatype, status, ierror)
                                                                                  11
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
                                                                                  12
                                                                                  13
    INTEGER, INTENT(IN) :: count
                                                                                  14
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  15
    TYPE(MPI_Status) :: status
                                                                                  16
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  17
MPI_File_read_all(fh, buf, count, datatype, status, ierror) !(_c)
                                                                                  18
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  19
    TYPE(*), DIMENSION(..) :: buf
                                                                                  20
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  21
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  22
    TYPE(MPI_Status) :: status
                                                                                  23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  24
                                                                                  25
MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
                                                                                  26
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  27
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  28
    INTEGER, INTENT(IN) :: count
                                                                                  29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  31
MPI_File_read_all_begin(fh, buf, count, datatype, ierror) !(_c)
                                                                                  32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  33
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  34
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  35
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  37
                                                                                  38
MPI_File_read_all_end(fh, buf, status, ierror)
                                                                                  39
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  40
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  41
    TYPE(MPI_Status) :: status
                                                                                  42
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  43
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
                                                                                  44
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  45
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  46
    TYPE(*), DIMENSION(..) :: buf
                                                                                  47
    INTEGER, INTENT(IN) :: count
                                                                                  48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
\mathbf{2}
         TYPE(MPI_Status) :: status
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror) !(_c)
5
         TYPE(MPI_File), INTENT(IN) :: fh
6
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
7
         TYPE(*), DIMENSION(..) :: buf
8
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
9
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
10
         TYPE(MPI_Status) :: status
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
14
         TYPE(MPI_File), INTENT(IN) :: fh
15
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
16
         TYPE(*), DIMENSION(..) :: buf
17
         INTEGER, INTENT(IN) :: count
18
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
19
         TYPE(MPI_Status) :: status
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
     MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
22
                  !(_c)
23
         TYPE(MPI_File), INTENT(IN) :: fh
24
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
25
         TYPE(*), DIMENSION(..) :: buf
26
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
27
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
28
         TYPE(MPI_Status) :: status
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
31
     MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
32
         TYPE(MPI_File), INTENT(IN) :: fh
33
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
34
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
35
         INTEGER, INTENT(IN) :: count
36
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
37
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
38
     MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror) !(_c)
39
         TYPE(MPI_File), INTENT(IN) :: fh
40
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
41
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
42
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
43
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
44
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
45
46
     MPI_File_read_at_all_end(fh, buf, status, ierror)
47
         TYPE(MPI_File), INTENT(IN) :: fh
48
```

```
TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  1
                                                                                  2
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  3
                                                                                  4
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
                                                                                  5
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  6
    TYPE(*), DIMENSION(..) :: buf
                                                                                  7
    INTEGER, INTENT(IN) :: count
                                                                                  8
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  9
    TYPE(MPI_Status) :: status
                                                                                  10
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  11
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror) !(_c)
                                                                                  12
                                                                                  13
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
                                                                                  14
                                                                                  15
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  16
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  17
    TYPE(MPI_Status) :: status
                                                                                  18
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  19
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
                                                                                  20
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  21
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  22
    INTEGER, INTENT(IN) :: count
                                                                                  23
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  24
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  25
                                                                                  26
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror) !(_c)
                                                                                  27
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  28
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  29
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  30
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  31
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  32
MPI_File_read_ordered_end(fh, buf, status, ierror)
                                                                                  33
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  34
    TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
                                                                                  35
    TYPE(MPI_Status) :: status
                                                                                  36
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  37
                                                                                  38
MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
                                                                                  39
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  40
    TYPE(*), DIMENSION(..) :: buf
                                                                                  41
    INTEGER, INTENT(IN) :: count
                                                                                  42
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  43
    TYPE(MPI_Status) :: status
                                                                                  44
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  45
MPI_File_read_shared(fh, buf, count, datatype, status, ierror) !(_c)
                                                                                  46
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  47
    TYPE(*), DIMENSION(..) :: buf
                                                                                  48
```

1001

```
1
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
\mathbf{2}
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
3
         TYPE(MPI_Status) :: status
4
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
5
     MPI_File_seek(fh, offset, whence, ierror)
6
         TYPE(MPI_File), INTENT(IN) :: fh
7
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
8
         INTEGER, INTENT(IN) :: whence
9
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
10
^{11}
     MPI_File_seek_shared(fh, offset, whence, ierror)
12
         TYPE(MPI_File), INTENT(IN) :: fh
13
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
14
         INTEGER, INTENT(IN) :: whence
15
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
16
    MPI_File_set_atomicity(fh, flag, ierror)
17
         TYPE(MPI_File), INTENT(IN) :: fh
18
         LOGICAL, INTENT(IN) :: flag
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
21
     MPI_File_set_info(fh, info, ierror)
22
         TYPE(MPI_File), INTENT(IN) :: fh
23
         TYPE(MPI_Info), INTENT(IN) :: info
^{24}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
25
    MPI_File_set_size(fh, size, ierror)
26
         TYPE(MPI_File), INTENT(IN) :: fh
27
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
31
         TYPE(MPI_File), INTENT(IN) :: fh
32
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: disp
33
         TYPE(MPI_Datatype), INTENT(IN) :: etype, filetype
34
         CHARACTER(LEN=*), INTENT(IN) :: datarep
35
         TYPE(MPI_Info), INTENT(IN) :: info
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_File_sync(fh, ierror)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
41
     MPI_File_write(fh, buf, count, datatype, status, ierror)
42
         TYPE(MPI_File), INTENT(IN) :: fh
43
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
44
         INTEGER, INTENT(IN) :: count
45
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
46
         TYPE(MPI_Status) :: status
47
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
MPI_File_write(fh, buf, count, datatype, status, ierror) !(_c)
                                                                                  1
                                                                                  2
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  3
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  4
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  5
                                                                                  6
    TYPE(MPI_Status) :: status
                                                                                  7
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  8
MPI_File_write_all(fh, buf, count, datatype, status, ierror)
                                                                                  9
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  10
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
                                                                                  11
    INTEGER, INTENT(IN) :: count
                                                                                  12
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  13
    TYPE(MPI_Status) :: status
                                                                                  14
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  15
                                                                                  16
MPI_File_write_all(fh, buf, count, datatype, status, ierror) !(_c)
                                                                                  17
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  18
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                  19
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  20
                                                                                  21
    TYPE(MPI_Status) :: status
                                                                                  22
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  23
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
                                                                                  24
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  25
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  26
    INTEGER, INTENT(IN) :: count
                                                                                  27
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  28
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  29
                                                                                  30
MPI_File_write_all_begin(fh, buf, count, datatype, ierror) !(_c)
                                                                                  31
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  32
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  33
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                  34
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  36
MPI_File_write_all_end(fh, buf, status, ierror)
                                                                                  37
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  38
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                  39
    TYPE(MPI_Status) :: status
                                                                                  40
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  41
                                                                                  42
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
                                                                                  43
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  44
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
                                                                                  45
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                  46
    INTEGER, INTENT(IN) :: count
                                                                                  47
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  48
```

```
1
         TYPE(MPI_Status) :: status
\mathbf{2}
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
3
     MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror) !(_c)
4
         TYPE(MPI_File), INTENT(IN) :: fh
5
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
6
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
7
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
8
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
9
         TYPE(MPI_Status) :: status
10
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
11
12
    MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
13
         TYPE(MPI_File), INTENT(IN) :: fh
14
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
15
         TYPE(*), DIMENSION(..), INTENT(IN) :: buf
16
         INTEGER, INTENT(IN) :: count
17
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
18
         TYPE(MPI_Status) :: status
19
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
20
     MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
21
                   !(_c)
22
         TYPE(MPI_File), INTENT(IN) :: fh
23
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
24
         TYPE(*), DIMENSION(...), INTENT(IN) :: buf
25
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
26
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
27
         TYPE(MPI_Status) :: status
28
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
29
30
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
^{31}
         TYPE(MPI_File), INTENT(IN) :: fh
32
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
33
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
34
         INTEGER, INTENT(IN) :: count
35
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
36
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
37
     MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror) !(_c)
38
         TYPE(MPI_File), INTENT(IN) :: fh
39
         INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
40
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
41
         INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
42
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_File_write_at_all_end(fh, buf, status, ierror)
46
         TYPE(MPI_File), INTENT(IN) :: fh
47
         TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
48
```

```
1
    TYPE(MPI_Status) :: status
                                                                                  2
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                  3
MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
                                                                                  4
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                  5
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
                                                                                  6
    INTEGER, INTENT(IN) :: count
                                                                                  7
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                  8
    TYPE(MPI_Status) :: status
                                                                                  9
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 10
                                                                                 11
MPI_File_write_ordered(fh, buf, count, datatype, status, ierror) !(_c)
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 12
                                                                                 13
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                 14
                                                                                 15
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 16
    TYPE(MPI_Status) :: status
                                                                                 17
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 18
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
                                                                                 19
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 20
    TYPE(*), DIMENSION(...), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 21
    INTEGER, INTENT(IN) :: count
                                                                                 22
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 23
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 24
                                                                                 25
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror) !(_c)
                                                                                 26
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 27
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 28
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                 29
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 31
MPI_File_write_ordered_end(fh, buf, status, ierror)
                                                                                 32
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 33
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
                                                                                 34
    TYPE(MPI_Status) :: status
                                                                                 35
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 36
                                                                                 37
MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
                                                                                 38
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 39
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                 40
    INTEGER, INTENT(IN) :: count
                                                                                 41
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
                                                                                 42
    TYPE(MPI_Status) :: status
                                                                                 43
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                 44
MPI_File_write_shared(fh, buf, count, datatype, status, ierror) !(_c)
                                                                                 45
    TYPE(MPI_File), INTENT(IN) :: fh
                                                                                 46
    TYPE(*), DIMENSION(...), INTENT(IN) :: buf
                                                                                 47
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
                                                                                 48
```

```
1
         TYPE(MPI_Datatype), INTENT(IN) :: datatype
\mathbf{2}
         TYPE(MPI_Status) :: status
3
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
4
     MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
5
                   dtype_file_extent_fn, extra_state, ierror)
6
         CHARACTER(LEN=*), INTENT(IN) :: datarep
7
         PROCEDURE(MPI_Datarep_conversion_function) :: read_conversion_fn,
8
                   write_conversion_fn
9
         PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
10
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
11
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
12
13
     MPI_Register_datarep_c(datarep, read_conversion_fn, write_conversion_fn,
14
                   dtype_file_extent_fn, extra_state, ierror) !(_c)
15
         CHARACTER(LEN=*), INTENT(IN) :: datarep
16
         PROCEDURE(MPI_Datarep_conversion_function_c) :: read_conversion_fn,
17
                   write_conversion_fn
18
         PROCEDURE(MPI_Datarep_extent_function) :: dtype_file_extent_fn
19
         INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
20
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
21
22
     A.4.13 Language Bindings Fortran 2008 Bindings
23
^{24}
     MPI_F_sync_reg(buf)
25
         TYPE(*), DIMENSION(...), ASYNCHRONOUS :: buf
26
    MPI_Status_f082f(f08_status, f_status, ierror)
27
         TYPE(MPI_Status), INTENT(IN) :: f08_status
28
         INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
29
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
30
^{31}
     MPI_Status_f2f08(f_status, f08_status, ierror)
32
         INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
33
         TYPE(MPI_Status), INTENT(OUT) :: f08_status
34
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
35
     MPI_Type_create_f90_complex(p, r, newtype, ierror)
36
         INTEGER, INTENT(IN) :: p, r
37
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
38
39
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
40
     MPI_Type_create_f90_integer(r, newtype, ierror)
41
         INTEGER, INTENT(IN) :: r
42
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
43
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
44
45
     MPI_Type_create_f90_real(p, r, newtype, ierror)
         INTEGER, INTENT(IN) :: p, r
46
47
         TYPE(MPI_Datatype), INTENT(OUT) :: newtype
         INTEGER, OPTIONAL, INTENT(OUT) :: ierror
48
```

```
1
MPI_Type_match_size(typeclass, size, datatype, ierror)
                                                                                      \mathbf{2}
    INTEGER, INTENT(IN) :: typeclass, size
                                                                                      3
    TYPE(MPI_Datatype), INTENT(OUT) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      4
                                                                                      5
                                                                                      6
A.4.14 Tools / Profiling Interface Fortran 2008 Bindings
                                                                                      7
                                                                                      8
MPI_Pcontrol(level)
                                                                                      9
    INTEGER, INTENT(IN) :: level
                                                                                      10
                                                                                      11
A.4.15 Deprecated Fortran 2008 Bindings
                                                                                      12
                                                                                      13
MPI_Info_get(info, key, valuelen, value, flag, ierror)
                                                                                      14
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                      15
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                      16
    INTEGER, INTENT(IN) :: valuelen
                                                                                      17
    CHARACTER(LEN=valuelen), INTENT(OUT) :: value
                                                                                      18
    LOGICAL, INTENT(OUT) :: flag
                                                                                      19
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      20
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
                                                                                      21
    TYPE(MPI_Info), INTENT(IN) :: info
                                                                                      22
    CHARACTER(LEN=*), INTENT(IN) :: key
                                                                                      23
    INTEGER, INTENT(OUT) :: valuelen
                                                                                      ^{24}
    LOGICAL, INTENT(OUT) :: flag
                                                                                      25
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      26
                                                                                      27
MPI_Sizeof(x, size, ierror)
                                                                                      28
    TYPE(*), DIMENSION(..) :: x
                                                                                      29
    INTEGER, INTENT(OUT) :: size
                                                                                      30
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
                                                                                      31
                                                                                      32
                                                                                      33
                                                                                      34
                                                                                      35
                                                                                      36
                                                                                      37
                                                                                      38
                                                                                      39
                                                                                      40
                                                                                      41
                                                                                      42
                                                                                      43
```

```
A.5
           Fortran Bindings with mpif.h or the mpi Module
1
\mathbf{2}
     A.5.1 Point-to-Point Communication Fortran Bindings
3
4
     MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
5
         <type> BUF(*)
6
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
7
     MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
8
9
         <type> BUF(*)
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
10
11
     MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
12
         <type> BUFFER(*)
13
         INTEGER SIZE, IERROR
14
15
    MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
16
         <type> BUFFER_ADDR(*)
17
         INTEGER SIZE, IERROR
18
     MPI_CANCEL(REQUEST, IERROR)
19
         INTEGER REQUEST, IERROR
20
21
     MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
22
         INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
23
    MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
24
         <type> BUF(*)
25
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
26
27
     MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
28
         INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
29
         LOGICAL FLAG
30
     MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
^{31}
         <type> BUF(*)
32
         INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR
33
34
     MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
35
         INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
36
         LOGICAL FLAG
37
     MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
38
         <type> BUF(*)
39
         INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
40
41
     MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
42
         <type> BUF(*)
43
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
44
     MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
45
         <type> BUF(*)
46
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
47
48
```

```
1008
```

MPI\_ISENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, 1 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, REQUEST, IERROR)  $\mathbf{2}$ 3 <type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, 4 5SOURCE, RECVTAG, COMM, REQUEST, IERROR 6 MPI\_ISENDRECV\_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, 7 COMM, REQUEST, IERROR) 8 <type> BUF(\*) 9 INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, REQUEST, 10IERROR 11 12MPI\_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 13 <type> BUF(\*) 14INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 15MPI\_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR) 16INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 1718 MPI\_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR) 19 <type> BUF(\*) 20 INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI\_STATUS\_SIZE), IERROR 21MPI\_PROBE(SOURCE, TAG, COMM, STATUS, IERROR) 22 INTEGER SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), IERROR 23  $^{24}$ MPI\_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR) 25<type> BUF(\*) 26INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI\_STATUS\_SIZE), 27IERROR 28MPI\_RECV\_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR) 29 <type> BUF(\*) 30 INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR 3132 MPI\_REQUEST\_FREE(REQUEST, IERROR) 33 INTEGER REQUEST, IERROR 34 MPI\_REQUEST\_GET\_STATUS (REQUEST, FLAG, STATUS, IERROR) 35 INTEGER REQUEST, STATUS(MPI\_STATUS\_SIZE), IERROR 36 LOGICAL FLAG 37 38 MPI\_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 39 <type> BUF(\*) 40INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 41 MPI\_RSEND\_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR) 42<type> BUF(\*) 43 INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR 4445MPI\_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR) 46<type> BUF(\*) 47INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR 48

```
1
    MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
\mathbf{2}
         <type> BUF(*)
3
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
4
     MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
\mathbf{5}
                   RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
6
         <type> SENDBUF(*), RECVBUF(*)
7
         INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
8
                    SOURCE, RECVTAG, COMM, STATUS(MPI STATUS SIZE), IERROR
9
10
     MPI_SENDRECV_REPLACE (BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
11
                   COMM, STATUS, IERROR)
12
         <type> BUF(*)
13
         INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
14
                   STATUS(MPI_STATUS_SIZE), IERROR
15
     MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
16
         <type> BUF(*)
17
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
18
19
     MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
20
         <type> BUF(*)
21
         INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
22
    MPI_START(REQUEST, IERROR)
23
         INTEGER REQUEST, IERROR
^{24}
25
    MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
26
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
27
     MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
28
         INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
29
         LOGICAL FLAG
30
31
     MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
32
         INTEGER STATUS(MPI_STATUS_SIZE), IERROR
33
         LOGICAL FLAG
34
     MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
35
         INTEGER COUNT, ARRAY_OF_REQUESTS(*),
36
                    ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
37
         LOGICAL FLAG
38
39
     MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS, IERROR)
40
         INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),
41
                   IERROR
42
         LOGICAL FLAG
43
    MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
44
                   ARRAY_OF_STATUSES, IERROR)
45
         INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*),
46
                    ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
47
48
```

MPI_WAIT(REQUEST, STATUS, IERROR) INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR	1 2
<pre>MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR) INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR</pre>	3 4 5
MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR) INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE),	6 7 8
<pre>IERROR MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,</pre>	9 10 11 12 13
A.5.2 Partitioned Communication Fortran Bindings	14 15 16 17
MPI_PARRIVED(REQUEST, PARTITION, FLAG, IERROR) INTEGER REQUEST, PARTITION, IERROR LOGICAL FLAG	18 19 20
MPI_PREADY(PARTITION, REQUEST, IERROR) INTEGER PARTITION, REQUEST, IERROR	21 22 23
<pre>MPI_PREADY_LIST(LENGTH, ARRAY_OF_PARTITIONS, REQUEST, IERROR) INTEGER LENGTH, ARRAY_OF_PARTITIONS(*), REQUEST, IERROR</pre>	24 25
MPI_PREADY_RANGE(PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR) INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR	26 27 28
<pre>MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, SOURCE, TAG, COMM, INFO,</pre>	29 30 31 32
INTEGER (KIND=MPI_COUNT_KIND) COUNT	33
<pre>MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR) <type> BUF(*)</type></pre>	34 35 36 37
INTEGER PARTITIONS, DATATYPE, DEST, TAG, COMM, INFO, REQUEST, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT	38 39 40
A.5.3 Datatypes Fortran Bindings	41
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP) INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP	42 43 44
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_DIFF(ADDR1, ADDR2) INTEGER(KIND=MPI_ADDRESS_KIND) ADDR1, ADDR2	45 $46$ $47$
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)	48

1 2 3	<type> LOCATION(*) INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS INTEGER IERROR</type>
4 5 6	MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
7 8 9	<pre>MPI_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) COUNT</pre>
10 11 12 13	<pre>MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)</pre>
14 15 16	MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, IERROR) CHARACTER*(*) DATAREP
17 18 19	<type> INBUF(*), OUTBUF(*) INTEGER INCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION</type>
20 21 22 23 24	<pre>MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)     CHARACTER*(*) DATAREP     INTEGER INCOUNT, DATATYPE, IERROR     INTEGER(KIND=MPI_ADDRESS_KIND) SIZE</pre>
25 26 27	MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR) INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
28 29 30	MPI_TYPE_COMMIT(DATATYPE, IERROR) INTEGER DATATYPE, IERROR
31 32	MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
33 34 35 36 37 38	<pre>MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,</pre>
39 40 41 42 43	<pre>MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>
43 44 45 46 47 48	<pre>MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,</pre>

MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)	1 2
INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE	3 4
<pre>MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR) INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR</pre>	5 6 7 8 9
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT	10 11 12
<pre>MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,</pre>	13 14 15 16 17 18
<pre>MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES, ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR) INTEGER NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*), ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR</pre>	19 20 21 22 23
MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR) INTEGER OLDTYPE, NEWTYPE, IERROR	23 24 25
MPI_TYPE_FREE(DATATYPE, IERROR) INTEGER DATATYPE, IERROR	26 27 28
<pre>MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES,</pre>	29 30 31 32 33 34
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR) INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR	35 36 37 38 39
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT	40 41 42
MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR) INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT	43 44 45 46
MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR) INTEGER DATATYPE, IERROR	40 47 48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT
\mathbf{2}
     MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
3
         INTEGER DATATYPE, IERROR
4
         INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
5
6
     MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
7
                   OLDTYPE, NEWTYPE, IERROR)
8
         INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
9
                    OLDTYPE, NEWTYPE, IERROR
10
     MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
11
         INTEGER DATATYPE, SIZE, IERROR
12
13
    MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
14
         INTEGER DATATYPE, IERROR
15
         INTEGER(KIND=MPI_COUNT_KIND) SIZE
16
    MPI TYPE VECTOR (COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
17
         INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
18
19
     MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
20
                   IERROR)
21
         <type> INBUF(*), OUTBUF(*)
22
         INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
23
     MPI_UNPACK_EXTERNAL (DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
^{24}
                   DATATYPE, IERROR)
25
         CHARACTER*(*) DATAREP
26
         <type> INBUF(*), OUTBUF(*)
27
         INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
28
         INTEGER OUTCOUNT, DATATYPE, IERROR
29
30
31
     A.5.4 Collective Communication Fortran Bindings
32
    MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
33
34
                   COMM, IERROR)
         <type> SENDBUF(*), RECVBUF(*)
35
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
36
37
     MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
38
                   RECVTYPE, COMM, INFO, REQUEST, IERROR)
39
         <type> SENDBUF(*), RECVBUF(*)
40
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
41
                    IERROR
42
     MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
43
44
                   RECVTYPE, COMM, IERROR)
45
         <type> SENDBUF(*), RECVBUF(*)
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
46
47
                    IERROR
48
```

MPI\_ALLGATHERV\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 1 DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 2 <type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 4 INFO, REQUEST, IERROR 5 6 MPI\_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 7 <type> SENDBUF(\*), RECVBUF(\*) 8 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 9 10 MPI\_ALLREDUCE\_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, 11 REQUEST, IERROR) <type> SENDBUF(\*), RECVBUF(\*) 1213 INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 14MPI\_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 15COMM, IERROR) 16<type> SENDBUF(\*), RECVBUF(\*) 17INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 18 19 MPI\_ALLTOALL\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 20RECVTYPE, COMM, INFO, REQUEST, IERROR) 21<type> SENDBUF(\*), RECVBUF(\*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 22 23IERROR 24MPI\_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, 25RDISPLS, RECVTYPE, COMM, IERROR) 26<type> SENDBUF(\*), RECVBUF(\*) 27INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*), 28RECVTYPE, COMM, IERROR 2930 MPI\_ALLTOALLV\_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 31RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR) 32 <type> SENDBUF(\*), RECVBUF(\*) 33 INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*), 34 RECVTYPE, COMM, INFO, REQUEST, IERROR 35 MPI\_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, 36 RDISPLS, RECVTYPES, COMM, IERROR) 37 <type> SENDBUF(\*), RECVBUF(\*) 38 INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPES(\*), RECVCOUNTS(\*), 39 RDISPLS(\*), RECVTYPES(\*), COMM, IERROR 4041 MPI\_ALLTOALLW\_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 42RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR) 43 <type> SENDBUF(\*), RECVBUF(\*) 44INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPES(\*), RECVCOUNTS(\*), 45RDISPLS(\*), RECVTYPES(\*), COMM, INFO, REQUEST, IERROR 46MPI\_BARRIER(COMM, IERROR) 47INTEGER COMM, IERROR 48

```
1
    MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
\mathbf{2}
         INTEGER COMM, INFO, REQUEST, IERROR
3
     MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
4
         <type> BUFFER(*)
5
         INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
6
7
     MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
8
         <type> BUFFER(*)
9
         INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR
10
     MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
11
         <type> SENDBUF(*), RECVBUF(*)
12
         INTEGER COUNT, DATATYPE, OP, COMM, IERROR
13
14
    MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
15
                   IERROR)
16
         <type> SENDBUF(*), RECVBUF(*)
17
         INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
18
    MPI_GATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
19
                   ROOT, COMM, IERROR)
20
         <type> SENDBUF(*), RECVBUF(*)
21
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
22
23
    MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
^{24}
                  ROOT, COMM, INFO, REQUEST, IERROR)
25
         <type> SENDBUF(*), RECVBUF(*)
26
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
27
                   REQUEST, IERROR
28
     MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
29
                   RECVTYPE, ROOT, COMM, IERROR)
30
         <type> SENDBUF(*), RECVBUF(*)
^{31}
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
32
                   COMM, IERROR
33
34
     MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
35
                   RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
36
         <type> SENDBUF(*), RECVBUF(*)
37
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
38
                   COMM, INFO, REQUEST, IERROR
39
     MPI_IALLGATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
40
                   COMM, REQUEST, IERROR)
41
         <type> SENDBUF(*), RECVBUF(*)
42
         INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
43
44
     MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
45
                   RECVTYPE, COMM, REQUEST, IERROR)
46
         <type> SENDBUF(*), RECVBUF(*)
47
48
```

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM, REQUEST, IERROR	1 2
<pre>MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,</pre>	3 4 5
INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR	6 7
<pre>MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGEP SENDCOUNT SENDTYPE PECYCOUNT PECYTYPE COMM PEQUEST IERPOP</type></pre>	8 9 10 11
<pre>MPI_IALLTOALLV(SENDBUF, SENDITIPE, RECVCUONI, RECVITPE, COMM, REQUEST, TERROR, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*), RECVTYPE, COMM, REQUEST, IERROR</type></pre>	12 13 14 15 16 17
<pre>MP1_TALLTOALLW(SENDBOF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBOF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*), RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR</type></pre>	<ol> <li>18</li> <li>19</li> <li>20</li> <li>21</li> <li>22</li> <li>23</li> </ol>
MPI_IBARRIER(COMM, REQUEST, IERROR)	24 25
<pre>MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)</pre>	26 27 28 29
	29 30 31 32
ROOT, COMM, REQUEST, IERROR) <type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, IERROR</type>	<ul> <li>33</li> <li>34</li> <li>35</li> <li>36</li> <li>37</li> <li>30</li> </ul>
<pre>MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,</pre>	<ol> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> </ol>
<pre>MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,</pre>	44 45 46 47 48

1 MPI\_IREDUCE\_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,  $\mathbf{2}$ REQUEST, IERROR) 3 <type> SENDBUF(\*), RECVBUF(\*) 4 INTEGER RECVCOUNTS(\*), DATATYPE, OP, COMM, REQUEST, IERROR 5MPI\_IREDUCE\_SCATTER\_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, 6 REQUEST, IERROR) 7 <type> SENDBUF(\*), RECVBUF(\*) 8 INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR 9 10MPI\_ISCAN (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR) 11 <type> SENDBUF(\*), RECVBUF(\*) 12INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR 13MPI\_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 14 ROOT, COMM, REQUEST, IERROR) 15<type> SENDBUF(\*), RECVBUF(\*) 16INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST, 17IERROR 18 19MPI\_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 20RECVTYPE, ROOT, COMM, REQUEST, IERROR) 21<type> SENDBUF(\*), RECVBUF(\*) 22INTEGER SENDCOUNTS(\*), DISPLS(\*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 23COMM, REQUEST, IERROR 24MPI\_OP\_COMMUTATIVE(OP, COMMUTE, IERROR) 25INTEGER OP, IERROR 26LOGICAL COMMUTE 2728MPI\_OP\_CREATE(USER\_FN, COMMUTE, OP, IERROR)  $^{29}$ EXTERNAL USER\_FN 30 LOGICAL COMMUTE  $^{31}$ INTEGER OP, IERROR 32MPI\_OP\_FREE(OP, IERROR) 33 INTEGER OP, IERROR 3435 MPI\_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR) 36 <type> SENDBUF(\*), RECVBUF(\*) 37 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR 38 MPI\_REDUCE\_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO, 39 REQUEST, IERROR) 40<type> SENDBUF(\*), RECVBUF(\*) 41 INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR 4243MPI\_REDUCE\_LOCAL (INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR) 44<type> INBUF(\*), INOUTBUF(\*) 45INTEGER COUNT, DATATYPE, OP, IERROR 4647MPI\_REDUCE\_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR) 48

1 <type> SENDBUF(\*), RECVBUF(\*) INTEGER RECVCOUNTS(\*), DATATYPE, OP, COMM, IERROR 2 3 MPI\_REDUCE\_SCATTER\_BLOCK (SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM, IERROR) 5<type> SENDBUF(\*), RECVBUF(\*) 6 INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR 7 8 MPI\_REDUCE\_SCATTER\_BLOCK\_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, 9 COMM, INFO, REQUEST, IERROR) 10<type> SENDBUF(\*), RECVBUF(\*) 11 INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 12MPI\_REDUCE\_SCATTER\_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, 13 INFO, REQUEST, IERROR) 14<type> SENDBUF(\*), RECVBUF(\*) 15INTEGER RECVCOUNTS(\*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR 1617MPI\_SCAN (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR) 18 <type> SENDBUF(\*), RECVBUF(\*) 19 INTEGER COUNT, DATATYPE, OP, COMM, IERROR 20MPI\_SCAN\_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST, 21IERROR) 22 <type> SENDBUF(\*), RECVBUF(\*) 23INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR 2425MPI\_SCATTER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, 26ROOT, COMM, IERROR) 27<type> SENDBUF(\*), RECVBUF(\*) 28INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR 29MPI\_SCATTER\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 30 RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 31<type> SENDBUF(\*), RECVBUF(\*) 32 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, 33 REQUEST, IERROR 34 35 MPI\_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, 36 RECVTYPE, ROOT, COMM, IERROR) 37 <type> SENDBUF(\*), RECVBUF(\*) 38 INTEGER SENDCOUNTS(\*), DISPLS(\*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 39 COMM, IERROR 40MPI\_SCATTERV\_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, 41 RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR) 42<type> SENDBUF(\*), RECVBUF(\*) 43 INTEGER SENDCOUNTS(\*), DISPLS(\*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, 44 COMM, INFO, REQUEST, IERROR 454647

48

```
1
     A.5.5 Groups, Contexts, Communicators, and Caching Fortran Bindings
\mathbf{2}
     MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
3
         INTEGER COMM1, COMM2, RESULT, IERROR
4
\mathbf{5}
     MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
6
         INTEGER COMM, GROUP, NEWCOMM, IERROR
7
     MPI_COMM_CREATE_FROM_GROUP(GROUP, STRINGTAG, INFO, ERRHANDLER, NEWCOMM,
8
                   IERROR)
9
         INTEGER GROUP, INFO, ERRHANDLER, NEWCOMM, IERROR
10
         CHARACTER*(*) STRINGTAG
11
12
     MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
13
         INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
14
     MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
15
                   EXTRA_STATE, IERROR)
16
         EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
17
         INTEGER COMM_KEYVAL, IERROR
18
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
19
20
     MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
21
         INTEGER COMM, COMM_KEYVAL, IERROR
22
     MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
23
         INTEGER COMM, NEWCOMM, IERROR
^{24}
25
     MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
26
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
27
         INTEGER OLDCOMM, COMM_KEYVAL, IERROR
28
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
29
                    ATTRIBUTE_VAL_OUT
30
         LOGICAL FLAG
^{31}
     MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
32
         INTEGER COMM, INFO, NEWCOMM, IERROR
33
34
     MPI_COMM_FREE(COMM, IERROR)
35
         INTEGER COMM, IERROR
36
     MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
37
         INTEGER COMM_KEYVAL, IERROR
38
39
     MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
40
         INTEGER COMM, COMM_KEYVAL, IERROR
41
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
42
         LOGICAL FLAG
43
     MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
44
         INTEGER COMM, INFO_USED, IERROR
45
46
     MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
47
         INTEGER COMM, RESULTLEN, IERROR
48
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 102	21
CHARACTER*(*) COMM_NAME	
MPI_COMM_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR	
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR) INTEGER COMM, NEWCOMM, REQUEST, IERROR	
MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR) INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR	
<pre>MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,</pre>	
MPI_COMM_NULL_DELETE_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR) INTEGER COMM, COMM_KEYVAL, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	
MPI_COMM_RANK(COMM, RANK, IERROR) INTEGER COMM, RANK, IERROR	
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR) INTEGER COMM, GROUP, IERROR	
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR	

- MPI\_COMM\_SET\_ATTR(COMM, COMM\_KEYVAL, ATTRIBUTE\_VAL, IERROR) INTEGER COMM, COMM\_KEYVAL, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) ATTRIBUTE\_VAL
- MPI\_COMM\_SET\_INFO(COMM, INFO, IERROR) INTEGER COMM, INFO, IERROR
- MPI\_COMM\_SET\_NAME(COMM, COMM\_NAME, IERROR) INTEGER COMM, IERROR CHARACTER\*(\*) COMM\_NAME
- MPI\_COMM\_SIZE(COMM, SIZE, IERROR) INTEGER COMM, SIZE, IERROR
- MPI\_COMM\_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR) INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
- MPI\_COMM\_SPLIT\_TYPE(COMM, SPLIT\_TYPE, KEY, INFO, NEWCOMM, IERROR) INTEGER COMM, SPLIT\_TYPE, KEY, INFO, NEWCOMM, IERROR

## MPI\_COMM\_TEST\_INTER(COMM, FLAG, IERROR) INTEGER COMM, IERROR LOGICAL FLAG

1 MPI\_GROUP\_COMPARE(GROUP1, GROUP2, RESULT, IERROR)  $\mathbf{2}$ INTEGER GROUP1, GROUP2, RESULT, IERROR 3 MPI\_GROUP\_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR) 4 INTEGER GROUP1, GROUP2, NEWGROUP, IERROR 56 MPI\_GROUP\_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR) 7 INTEGER GROUP, N, RANKS(\*), NEWGROUP, IERROR 8 MPI\_GROUP\_FREE(GROUP, IERROR) 9 INTEGER GROUP, IERROR 10  $^{11}$ MPI\_GROUP\_FROM\_SESSION\_PSET(SESSION, PSET\_NAME, NEWGROUP, IERROR) 12INTEGER SESSION, NEWGROUP, IERROR 13CHARACTER\*(\*) PSET\_NAME 14MPI\_GROUP\_INCL(GROUP, N, RANKS, NEWGROUP, IERROR) 15INTEGER GROUP, N, RANKS(\*), NEWGROUP, IERROR 1617MPI\_GROUP\_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR) 18 INTEGER GROUP1, GROUP2, NEWGROUP, IERROR 19MPI\_GROUP\_RANGE\_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR) 20INTEGER GROUP, N, RANGES(3, \*), NEWGROUP, IERROR 2122MPI\_GROUP\_RANGE\_INCL(GROUP, N, RANGES, NEWGROUP, IERROR) 23INTEGER GROUP, N, RANGES(3, \*), NEWGROUP, IERROR  $^{24}$ MPI\_GROUP\_RANK(GROUP, RANK, IERROR) 25INTEGER GROUP, RANK, IERROR 2627MPI\_GROUP\_SIZE(GROUP, SIZE, IERROR) 28INTEGER GROUP, SIZE, IERROR 29MPI\_GROUP\_TRANSLATE\_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR) 30 INTEGER GROUP1, N, RANKS1(\*), GROUP2, RANKS2(\*), IERROR  $^{31}$ 32 MPI\_GROUP\_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) 33 INTEGER GROUP1, GROUP2, NEWGROUP, IERROR 34 MPI\_INTERCOMM\_CREATE(LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE LEADER, 35 TAG, NEWINTERCOMM, IERROR) 36 37 INTEGER LOCAL\_COMM, LOCAL\_LEADER, PEER\_COMM, REMOTE\_LEADER, TAG, 38 NEWINTERCOMM, IERROR 39 MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS(LOCAL\_GROUP, LOCAL\_LEADER, REMOTE\_GROUP, 40REMOTE\_LEADER, STRINGTAG, INFO, ERRHANDLER, NEWINTERCOMM, 41 IERROR) 42INTEGER LOCAL\_GROUP, LOCAL\_LEADER, REMOTE\_GROUP, REMOTE\_LEADER, INFO, 43 ERRHANDLER, NEWINTERCOMM, IERROR 44CHARACTER\*(\*) STRINGTAG 4546MPI\_INTERCOMM\_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR) 47INTEGER INTERCOMM, NEWINTRACOMM, IERROR 48LOGICAL HIGH

MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,	1
EXTRA_STATE, IERROR) EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN	3
INTEGER TYPE_KEYVAL, IERROR	4
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	5
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)	6
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	7 8
MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	9
ATTRIBUTE_VAL_OUT, FLAG, IERROR)	10
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	11
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	12
ATTRIBUTE_VAL_OUT	13 14
LOGICAL FLAG	14
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)	16
INTEGER TYPE_KEYVAL, IERROR	17
MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)	18
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	19 20
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	20 21
LOGICAL FLAG	22
MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)	23
INTEGER DATATYPE, RESULTLEN, IERROR	24
CHARACTER*(*) TYPE_NAME	25
MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	26 27
ATTRIBUTE_VAL_OUT, FLAG, IERROR) INTEGER OLDTYPE, TYPE_KEYVAL, IERROR	28
INTEGER OLDITPE, TIPE_KEIVAL, TERROR INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,	29
ATTRIBUTE_VAL_OUT	30
LOGICAL FLAG	31
MPI_TYPE_NULL_DELETE_FN(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,	32 33
IERROR)	34
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	35
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE	36
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)	37
INTEGER DATATYPE, TYPE_KEYVAL, IERROR	38 39
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL	40
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)	41
INTEGER DATATYPE, IERROR	42
CHARACTER*(*) TYPE_NAME	43
MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,	44 45
EXTRA_STATE, IERROR)	45 46
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN	47
INTEGER WIN_KEYVAL, IERROR	48

```
1
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
\mathbf{2}
     MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
3
         INTEGER WIN, WIN_KEYVAL, IERROR
4
\mathbf{5}
     MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
6
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
7
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
8
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
9
                    ATTRIBUTE_VAL_OUT
10
         LOGICAL FLAG
11
     MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
12
         INTEGER WIN_KEYVAL, IERROR
13
14
     MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
15
         INTEGER WIN, WIN_KEYVAL, IERROR
16
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
17
         LOGICAL FLAG
18
     MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
19
         INTEGER WIN, RESULTLEN, IERROR
20
         CHARACTER*(*) WIN_NAME
21
22
     MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
23
                   ATTRIBUTE_VAL_OUT, FLAG, IERROR)
^{24}
         INTEGER OLDWIN, WIN_KEYVAL, IERROR
25
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
26
                    ATTRIBUTE_VAL_OUT
27
         LOGICAL FLAG
28
     MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
29
         INTEGER WIN, WIN_KEYVAL, IERROR
30
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
^{31}
32
     MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
33
         INTEGER WIN, WIN_KEYVAL, IERROR
34
         INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
35
     MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
36
         INTEGER WIN, IERROR
37
         CHARACTER*(*) WIN_NAME
38
39
40
     A.5.6 Process Topologies Fortran Bindings
41
42
     MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
43
         INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
44
     MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
45
         INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
46
         LOGICAL PERIODS(*), REORDER
47
48
     MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE 102	25
<pre>INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR LOGICAL PERIODS(*)</pre>	1 2
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR) INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR LOGICAL PERIODS(*)	3 4 5 6
MPI_CART_RANK(COMM, COORDS, RANK, IERROR) INTEGER COMM, COORDS(*), RANK, IERROR	7 8
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR) INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR	9 10 11
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR) INTEGER COMM, NEWCOMM, IERROR LOGICAL REMAIN_DIMS(*)	12 13 14
MPI_CARTDIM_GET(COMM, NDIMS, IERROR) INTEGER COMM, NDIMS, IERROR	15 16 17
<pre>MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR) INTEGER NNODES, NDIMS, DIMS(*), IERROR</pre>	18 19 20
<pre>MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS INFO, REORDER, COMM_DIST_GRAPH, IERROR) INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*), WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER</pre>	
<pre>MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS, OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER, COMM_DIST_GRAPH, IERROR) INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR LOGICAL REORDER</pre>	27 28 29 30 31 32
<pre>MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS, MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR) INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE, DESTINATIONS(*), DESTWEIGHTS(*), IERROR</pre>	33 34 35 36 37
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR INTEGER COMM, INDEGREE, OUTDEGREE, IERROR LOGICAL WEIGHTED	2) 38 39 40 41
<pre>MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, IERROR) INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR LOGICAL REORDER</pre>	41 42 43 44 45
<pre>MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR) INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR</pre>	46 47 48

1 MPI\_GRAPH\_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)  $\mathbf{2}$ INTEGER COMM, NNODES, INDEX(\*), EDGES(\*), NEWRANK, IERROR 3 MPI\_GRAPH\_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR) 4 INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(\*), IERROR 56 MPI\_GRAPH\_NEIGHBORS\_COUNT(COMM, RANK, NNEIGHBORS, IERROR) 7 INTEGER COMM, RANK, NNEIGHBORS, IERROR 8 MPI\_GRAPHDIMS\_GET(COMM, NNODES, NEDGES, IERROR) 9 INTEGER COMM, NNODES, NEDGES, IERROR 10  $^{11}$ MPI\_INEIGHBOR\_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 12RECVTYPE, COMM, REQUEST, IERROR) 13<type> SENDBUF(\*), RECVBUF(\*) 14INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 15MPI\_INEIGHBOR\_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, 16DISPLS, RECVTYPE, COMM, REQUEST, IERROR) 17<type> SENDBUF(\*), RECVBUF(\*) 18 INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(\*), DISPLS(\*), RECVTYPE, COMM, 19REQUEST, IERROR 2021MPI\_INEIGHBOR\_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 22RECVTYPE, COMM, REQUEST, IERROR) 23<type> SENDBUF(\*), RECVBUF(\*)  $^{24}$ INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR 25MPI\_INEIGHBOR\_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, 26RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR) 27<type> SENDBUF(\*), RECVBUF(\*) 28INTEGER SENDCOUNTS(\*), SDISPLS(\*), SENDTYPE, RECVCOUNTS(\*), RDISPLS(\*), 29RECVTYPE, COMM, REQUEST, IERROR 30 31MPI\_INEIGHBOR\_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, 32 RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR) 33 <type> SENDBUF(\*), RECVBUF(\*) 34 INTEGER SENDCOUNTS(\*), SENDTYPES(\*), RECVCOUNTS(\*), RECVTYPES(\*), COMM, 35 REQUEST, IERROR 36 INTEGER(KIND=MPI\_ADDRESS\_KIND) SDISPLS(\*), RDISPLS(\*) 37 MPI\_NEIGHBOR\_ALLGATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, 38 RECVTYPE, COMM, IERROR) 39 <type> SENDBUF(\*), RECVBUF(\*) 40INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR 41 42MPI\_NEIGHBOR\_ALLGATHER\_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, 43 RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR) 44<type> SENDBUF(\*), RECVBUF(\*) 45INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, 46IERROR 4748

MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,	1
DISPLS, RECVTYPE, COMM, IERROR)	2
<pre><type> SENDBUF(*), RECVBUF(*) </type></pre>	3
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	4 5
IERROR	э 6
MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,	7
RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)	8
<type> SENDBUF(*), RECVBUF(*)</type>	9
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,	10
INFO, REQUEST, IERROR	11
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,	12
RECVTYPE, COMM, IERROR)	13
<type> SENDBUF(*), RECVBUF(*)</type>	14
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR	15
	16
MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,	17
RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)	18
<pre><type> SENDBUF(*), RECVBUF(*) </type></pre>	19
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,	20
IERROR	21
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,	22
RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)	23
<type> SENDBUF(*), RECVBUF(*)</type>	24
<pre>INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),</pre>	25
RECVTYPE, COMM, IERROR	26
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,	27 28
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,	28 29
IERROR)	30
<type> SENDBUF(*), RECVBUF(*)</type>	31
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),	32
RECVTYPE, COMM, INFO, REQUEST, IERROR	33
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,	34
RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)	35
<pre><type> SENDBUF(*), RECVBUF(*)</type></pre>	36
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,	37
IERROR	38
INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)	39
	40
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,	41
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,	42
	43
<pre><type> SENDBUF(*), RECVBUF(*) </type></pre>	44
INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,	45
INFO, REQUEST, IERROR	46
<pre>INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)</pre>	47
	48

```
1
     MPI_TOPO_TEST(COMM, STATUS, IERROR)
\mathbf{2}
         INTEGER COMM, STATUS, IERROR
3
4
     A.5.7 MPI Environmental Management Fortran Bindings
5
6
     DOUBLE PRECISION MPI_WTICK()
7
     DOUBLE PRECISION MPI_WTIME()
8
9
     MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
10
         INTEGER ERRORCLASS, IERROR
11
     MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
12
         INTEGER ERRORCLASS, ERRORCODE, IERROR
13
14
     MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
15
         INTEGER ERRORCODE, IERROR
16
         CHARACTER*(*) STRING
17
     MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
18
19
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
         INTEGER INFO, IERROR
20
21
     If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
22
       INTERFACE MPI_ALLOC_MEM
23
         SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
^{24}
           IMPORT :: MPI_ADDRESS_KIND
25
           INTEGER :: INFO, IERROR
26
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
27
         END SUBROUTINE
28
         SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
^{29}
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
30
           IMPORT :: MPI_ADDRESS_KIND
^{31}
           INTEGER :: INFO, IERROR
32
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
33
           TYPE(C_PTR) :: BASEPTR
34
         END SUBROUTINE
35
       END INTERFACE
36
37
     MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
38
         INTEGER COMM, ERRORCODE, IERROR
39
     MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
40
         EXTERNAL COMM_ERRHANDLER_FN
41
         INTEGER ERRHANDLER, IERROR
42
     MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
43
44
         INTEGER COMM, ERRHANDLER, IERROR
45
     MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
46
         INTEGER COMM, ERRHANDLER, IERROR
47
48
     MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
```

INTEGER ERRHANDLER, IERROR	1
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)	2
INTEGER ERRORCODE, ERRORCLASS, IERROR	3 4
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)	5
INTEGER ERRORCODE, RESULTLEN, IERROR	6
CHARACTER*(*) STRING	7
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)	8
INTEGER FH, ERRORCODE, IERROR	9 10
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)	11
EXTERNAL FILE_ERRHANDLER_FN	12
INTEGER ERRHANDLER, IERROR	13
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	14 15
INTEGER FILE, ERRHANDLER, IERROR	16
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)	17
INTEGER FILE, ERRHANDLER, IERROR	18
	19
MPI_FREE_MEM(BASE, IERROR)	20
<type> BASE(*)</type>	21
INTEGER IERROR	22
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)	23 24
CHARACTER*(*) VERSION	24 25
INTEGER RESULTLEN, IERROR	26
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)	27
CHARACTER*(*) NAME	28
INTEGER RESULTLEN, IERROR	29
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)	30
INTEGER VERSION, SUBVERSION, IERROR	31
	32
MPI_SESSION_CALL_ERRHANDLER(SESSION, ERRORCODE, IERROR)	33
INTEGER SESSION, ERRORCODE, IERROR	34 35
MPI_SESSION_CREATE_ERRHANDLER(SESSION_ERRHANDLER_FN, ERRHANDLER, IERROR)	36
EXTERNAL SESSION_ERRHANDLER_FN	37
INTEGER ERRHANDLER, IERROR	38
MPI_SESSION_GET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)	39
INTEGER SESSION, ERRHANDLER, IERROR	40
MPI_SESSION_SET_ERRHANDLER(SESSION, ERRHANDLER, IERROR)	41
INTEGER SESSION, ERRHANDLER, IERROR	42
	43
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)	44
INTEGER WIN, ERRORCODE, IERROR	45
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)	46 47
EXTERNAL WIN_ERRHANDLER_FN	48

```
1
         INTEGER ERRHANDLER, IERROR
\mathbf{2}
     MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
3
         INTEGER WIN, ERRHANDLER, IERROR
4
\mathbf{5}
     MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
6
         INTEGER WIN, ERRHANDLER, IERROR
7
8
     A.5.8 The Info Object Fortran Bindings
9
10
     MPI_INFO_CREATE(INFO, IERROR)
^{11}
         INTEGER INFO, IERROR
12
     MPI_INFO_CREATE_ENV(INFO, IERROR)
13
         INTEGER INFO, IERROR
14
15
     MPI_INFO_DELETE(INFO, KEY, IERROR)
16
         INTEGER INFO, IERROR
17
         CHARACTER*(*) KEY
18
     MPI_INFO_DUP(INFO, NEWINFO, IERROR)
19
         INTEGER INFO, NEWINFO, IERROR
20
21
     MPI_INFO_FREE(INFO, IERROR)
22
         INTEGER INFO, IERROR
23
^{24}
     MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
         INTEGER INFO, NKEYS, IERROR
25
26
     MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
27
         INTEGER INFO, N, IERROR
28
         CHARACTER*(*) KEY
29
30
     MPI_INFO_GET_STRING(INFO, KEY, BUFLEN, VALUE, FLAG, IERROR)
^{31}
         INTEGER INFO, BUFLEN, IERROR
32
         CHARACTER*(*) KEY, VALUE
33
         LOGICAL FLAG
34
     MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
35
         INTEGER INFO, IERROR
36
         CHARACTER*(*) KEY, VALUE
37
38
39
     A.5.9 Process Creation and Management Fortran Bindings
40
     MPI_ABORT(COMM, ERRORCODE, IERROR)
41
         INTEGER COMM, ERRORCODE, IERROR
42
43
     MPI_CLOSE_PORT(PORT_NAME, IERROR)
44
         CHARACTER*(*) PORT_NAME
45
         INTEGER IERROR
46
     MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
47
         CHARACTER*(*) PORT_NAME
48
```

INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR 1 2 MPI\_COMM\_CONNECT(PORT\_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR) 3 CHARACTER\*(\*) PORT\_NAME 4 INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR 5 6 MPI\_COMM\_DISCONNECT(COMM, IERROR) INTEGER COMM, IERROR 7 MPI\_COMM\_GET\_PARENT(PARENT, IERROR) 9 INTEGER PARENT, IERROR 10 11 MPI\_COMM\_JOIN(FD, INTERCOMM, IERROR) INTEGER FD, INTERCOMM, IERROR 1213 MPI\_COMM\_SPAWN (COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, 14ARRAY\_OF\_ERRCODES, IERROR) 15CHARACTER\*(\*) COMMAND, ARGV(\*) 16INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY\_OF\_ERRCODES(\*), 17IERROR 18 19MPI\_COMM\_SPAWN\_MULTIPLE(COUNT, ARRAY\_OF\_COMMANDS, ARRAY\_OF\_ARGV, 20ARRAY\_OF\_MAXPROCS, ARRAY\_OF\_INFO, ROOT, COMM, INTERCOMM, 21ARRAY\_OF\_ERRCODES, IERROR) 22 INTEGER COUNT, ARRAY\_OF\_MAXPROCS(\*), ARRAY\_OF\_INFO(\*), ROOT, COMM, 23INTERCOMM, ARRAY\_OF\_ERRCODES(\*), IERROR 24CHARACTER\*(\*) ARRAY\_OF\_COMMANDS(\*), ARRAY\_OF\_ARGV(COUNT, \*) 25MPI\_FINALIZE(IERROR) 26INTEGER IERROR 2728 MPI\_FINALIZED(FLAG, IERROR) 29LOGICAL FLAG 30 INTEGER IERROR 31MPI\_INIT(IERROR) 32 INTEGER IERROR 33 34 MPI\_INIT\_THREAD(REQUIRED, PROVIDED, IERROR) 35 INTEGER REQUIRED, PROVIDED, IERROR 36 MPI\_INITIALIZED(FLAG, IERROR) 37 LOGICAL FLAG 38 INTEGER IERROR 39 40 MPI\_IS\_THREAD\_MAIN(FLAG, IERROR) 41 LOGICAL FLAG 42INTEGER IERROR 43 MPI\_LOOKUP\_NAME(SERVICE\_NAME, INFO, PORT\_NAME, IERROR) 44CHARACTER\*(\*) SERVICE\_NAME, PORT\_NAME 45INTEGER INFO, IERROR 4647MPI\_OPEN\_PORT(INFO, PORT\_NAME, IERROR) 48

```
1
         INTEGER INFO, IERROR
\mathbf{2}
         CHARACTER*(*) PORT_NAME
3
     MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
4
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
5
         INTEGER INFO, IERROR
6
\overline{7}
    MPI_QUERY_THREAD(PROVIDED, IERROR)
8
         INTEGER PROVIDED, IERROR
9
    MPI_SESSION_FINALIZE(SESSION, IERROR)
10
         INTEGER SESSION, IERROR
11
12
    MPI_SESSION_GET_INFO(SESSION, INFO_USED, IERROR)
13
         INTEGER SESSION, INFO_USED, IERROR
14
     MPI_SESSION_GET_NTH_PSET(SESSION, INFO, N, PSET_LEN, PSET_NAME, IERROR)
15
         INTEGER SESSION, INFO, N, PSET_LEN, IERROR
16
         CHARACTER*(*) PSET_NAME
17
18
    MPI_SESSION_GET_NUM_PSETS(SESSION, INFO, NPSET_NAMES, IERROR)
19
         INTEGER SESSION, INFO, NPSET_NAMES, IERROR
20
     MPI_SESSION_GET_PSET_INFO(SESSION, PSET_NAME, INFO, IERROR)
21
         INTEGER SESSION, INFO, IERROR
22
         CHARACTER*(*) PSET_NAME
23
^{24}
     MPI_SESSION_INIT(INFO, ERRHANDLER, SESSION, IERROR)
25
         INTEGER INFO, ERRHANDLER, SESSION, IERROR
26
     MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
27
         CHARACTER*(*) SERVICE_NAME, PORT_NAME
28
         INTEGER INFO, IERROR
29
30
31
     A.5.10 One-Sided Communications Fortran Bindings
32
33
     MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
34
                   TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
35
         <type> ORIGIN_ADDR(*)
         INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
36
37
                    TARGET_DATATYPE, OP, WIN, IERROR
38
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
39
     MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
40
                   TARGET_RANK, TARGET_DISP, WIN, IERROR)
41
         <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
42
         INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
43
         INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
44
45
     MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
46
                   TARGET_DISP, OP, WIN, IERROR)
47
         <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
48
```

INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	$\frac{1}{2}$
	3
MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	4
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)	5
<pre><type> ORIGIN_ADDR(*) INTEGED ORIGIN_COUNT_ORIGIN_DATATYDETARGET_DANKTARGET_COUNT_</type></pre>	6
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	7
TARGET_DATATYPE, WIN, IERROR	8
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	9
MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,	10
RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,	11
TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)	12
<type> ORIGIN_ADDR(*), RESULT_ADDR(*)</type>	13
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,	14
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR	15
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	16
MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	17
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)	18
<pre><type> ORIGIN_ADDR(*)</type></pre>	19
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	20
TARGET_DATATYPE, WIN, IERROR	21
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	22
	23
MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	24
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	25 26
IERROR)	20 27
<type> ORIGIN_ADDR(*)</type>	21
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	20
TARGET_DATATYPE, OP, WIN, REQUEST, IERROR	30
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	31
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,	32
TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,	33
IERROR)	34
<type> ORIGIN_ADDR(*)</type>	35
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,	36
TARGET_DATATYPE, WIN, REQUEST, IERROR	37
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	38
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,	39
RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,	40
TARGET_DISP, TARGET_COUNT, TARGET_DATATIVE, TARGET_TANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	41
IARGEI_DISF, IARGEI_COUNI, IARGEI_DAIAIIPE, OF, WIN, REQUESI, IERROR)	42
<pre>type&gt; ORIGIN_ADDR(*), RESULT_ADDR(*)</pre>	43
INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,	44
TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,	45
IERROR	46
INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP	47
	48

1 MPI\_RPUT(ORIGIN\_ADDR, ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK,  $\mathbf{2}$ TARGET\_DISP, TARGET\_COUNT, TARGET\_DATATYPE, WIN, REQUEST, 3 IERROR) 4 <type> ORIGIN\_ADDR(\*) 5INTEGER ORIGIN\_COUNT, ORIGIN\_DATATYPE, TARGET\_RANK, TARGET\_COUNT, 6 TARGET\_DATATYPE, WIN, REQUEST, IERROR 7 INTEGER(KIND=MPI\_ADDRESS\_KIND) TARGET\_DISP 8 MPI\_WIN\_ALLOCATE(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 9 INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR 10 INTEGER DISP\_UNIT, INFO, COMM, WIN, IERROR 11 If the Fortran compiler provides TYPE(C\_PTR), then overloaded by: 12INTERFACE MPI\_WIN\_ALLOCATE 13 SUBROUTINE MPI\_WIN\_ALLOCATE(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, & 14WIN, IERROR) 15IMPORT :: MPI ADDRESS KIND 16INTEGER :: DISP\_UNIT, INFO, COMM, WIN, IERROR 17 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE, BASEPTR 18 19 END SUBROUTINE SUBROUTINE MPI\_WIN\_ALLOCATE\_CPTR(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, & 20WIN, IERROR) 21USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR 2223IMPORT :: MPI\_ADDRESS\_KIND INTEGER :: DISP\_UNIT, INFO, COMM, WIN, IERROR  $^{24}$ INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE 2526TYPE(C\_PTR) :: BASEPTR 27END SUBROUTINE END INTERFACE 2829 MPI\_WIN\_ALLOCATE\_SHARED(SIZE, DISP\_UNIT, INFO, COMM, BASEPTR, WIN, IERROR) 30 INTEGER(KIND=MPI\_ADDRESS\_KIND) SIZE, BASEPTR  $^{31}$ INTEGER DISP\_UNIT, INFO, COMM, WIN, IERROR 32 If the Fortran compiler provides TYPE(C\_PTR), then overloaded by: 33 34INTERFACE MPI\_WIN\_ALLOCATE\_SHARED 35SUBROUTINE MPI\_WIN\_ALLOCATE\_SHARED(SIZE, DISP\_UNIT, INFO, COMM, & 36 BASEPTR, WIN, IERROR) 37 IMPORT :: MPI\_ADDRESS\_KIND 38 INTEGER :: DISP\_UNIT, INFO, COMM, WIN, IERROR 39 INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE, BASEPTR 40END SUBROUTINE 41 SUBROUTINE MPI\_WIN\_ALLOCATE\_SHARED\_CPTR(SIZE, DISP\_UNIT, INFO, COMM, & 42BASEPTR, WIN, IERROR) USE, INTRINSIC :: ISO\_C\_BINDING, ONLY : C\_PTR 4344IMPORT :: MPI\_ADDRESS\_KIND 45INTEGER :: DISP\_UNIT, INFO, COMM, WIN, IERROR INTEGER(KIND=MPI\_ADDRESS\_KIND) :: SIZE 4647TYPE(C\_PTR) :: BASEPTR 48 END SUBROUTINE

END INTERFACE	1
MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR) INTEGER WIN, IERROR	2 3 4
<type> BASE(*) INTEGER(KIND=MPI_ADDRESS_KIND) SIZE</type>	5
MPI_WIN_COMPLETE(WIN, IERROR)	7
INTEGER WIN, IERROR	9
<pre>MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)</pre>	10
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR	12
MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)	14
INTEGER INFO, COMM, WIN, IERROR	16
MPI_WIN_DETACH(WIN, BASE, IERROR) INTEGER WIN, IERROR	17
<type> BASE(*)</type>	19
MPI_WIN_FENCE(ASSERT, WIN, IERROR)	20
INTEGER ASSERT, WIN, IERROR	21 22
MPI_WIN_FLUSH(RANK, WIN, IERROR)	23
INTEGER RANK, WIN, IERROR	24
MPI_WIN_FLUSH_ALL(WIN, IERROR)	25
INTEGER WIN, IERROR	26
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)	27
INTEGER RANK, WIN, IERROR	28 29
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)	30
INTEGER WIN, IERROR	31 32
MPI_WIN_FREE(WIN, IERROR)	33
INTEGER WIN, IERROR	34
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)	35
INTEGER WIN, GROUP, IERROR	36
MET UTN CET INCO (UTN INCO UCED IEDDOD)	37
MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR) INTEGER WIN, INFO_USED, IERROR	39
	40
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)	41
INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR	42
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)	43
INTEGER ASSERT, WIN, IERROR	44 45
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)	45
INTEGER GROUP, ASSERT, WIN, IERROR	47
MPI_WIN_SET_INFO(WIN, INFO, IERROR)	48

```
1
         INTEGER WIN, INFO, IERROR
\mathbf{2}
     MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
3
         INTEGER WIN, RANK, DISP_UNIT, IERROR
4
         INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
5
     If the Fortran compiler provides TYPE(C_PTR), then overloaded by:
6
7
       INTERFACE MPI_WIN_SHARED_QUERY
         SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
8
9
               BASEPTR, IERROR)
           IMPORT :: MPI_ADDRESS_KIND
10
11
           INTEGER :: WIN, RANK, DISP_UNIT, IERROR
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
12
         END SUBROUTINE
13
         SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
14
               BASEPTR, IERROR)
15
           USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
16
           IMPORT :: MPI_ADDRESS_KIND
17
           INTEGER :: WIN, RANK, DISP_UNIT, IERROR
18
19
           INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
           TYPE(C_PTR) :: BASEPTR
20
21
         END SUBROUTINE
       END INTERFACE
22
23
     MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
24
         INTEGER GROUP, ASSERT, WIN, IERROR
25
26
     MPI_WIN_SYNC(WIN, IERROR)
27
         INTEGER WIN, IERROR
28
     MPI_WIN_TEST(WIN, FLAG, IERROR)
29
         INTEGER WIN, IERROR
30
         LOGICAL FLAG
^{31}
32
     MPI_WIN_UNLOCK(RANK, WIN, IERROR)
33
         INTEGER RANK, WIN, IERROR
34
     MPI_WIN_UNLOCK_ALL(WIN, IERROR)
35
         INTEGER WIN, IERROR
36
37
     MPI_WIN_WAIT(WIN, IERROR)
38
         INTEGER WIN, IERROR
39
40
     A.5.11 External Interfaces Fortran Bindings
41
42
     MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
43
         INTEGER REQUEST, IERROR
44
45
     MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
46
                   IERROR)
47
         EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
48
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

INTEGER REQUEST, IERROR	1
MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)	2
INTEGER STATUS(MPI_STATUS_SIZE), IERROR	3 4
LOGICAL FLAG	5
MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)	6
INTEGER STATUS (MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR	7
	8
MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR) INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR	9
INTEGER (KIND=MPI_COUNT_KIND) COUNT	10
	11 12
	12
A.5.12 I/O Fortran Bindings	14
MPI_CONVERSION_FN_NULL(USERBUF, DATATYPE, COUNT, FILEBUF, POSITION,	15
EXTRA_STATE, IERROR)	16
<type> USERBUF(*), FILEBUF(*)</type>	17
INTEGER DATATYPE, COUNT, IERROR	18
INTEGER(KIND=MPI_OFFSET_KIND) POSITION	19
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE	20 21
MPI_FILE_CLOSE(FH, IERROR)	21
INTEGER FH, IERROR	23
MPI_FILE_DELETE(FILENAME, INFO, IERROR)	24
CHARACTER*(*) FILENAME	25
INTEGER INFO, IERROR	26
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)	27
INTEGER FH, AMODE, IERROR	28
	29 30
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR) INTEGER FH, IERROR	30
LOGICAL FLAG	32
	33
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)	34
INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP	35
	36
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)	37
INTEGER FH, GROUP, IERROR	38 39
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)	40
INTEGER FH, INFO_USED, IERROR	41
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)	42
INTEGER FH, IERROR	43
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	44
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)	45
INTEGER FH, IERROR	46
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET	47 48

```
1
     MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
\mathbf{2}
         INTEGER FH, IERROR
3
         INTEGER(KIND=MPI_OFFSET_KIND) SIZE
4
     MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
\mathbf{5}
         INTEGER FH, DATATYPE, IERROR
6
         INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT
\overline{7}
8
     MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
9
         INTEGER FH, ETYPE, FILETYPE, IERROR
10
         INTEGER(KIND=MPI_OFFSET_KIND) DISP
11
         CHARACTER*(*) DATAREP
12
     MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
13
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
14
         <type> BUF(*)
15
16
     MPI_FILE_IREAD_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
17
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
18
         <type> BUF(*)
19
     MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
20
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
21
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
22
         <type> BUF(*)
23
^{24}
     MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
25
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
26
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
27
         <type> BUF(*)
28
     MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
29
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
30
         <type> BUF(*)
^{31}
32
     MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
33
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
34
         <type> BUF(*)
35
     MPI_FILE_IWRITE_ALL(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
36
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
37
         <type> BUF(*)
38
39
     MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
40
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
41
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
42
         <type> BUF(*)
43
     MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
44
         INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
45
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
46
47
         <type> BUF(*)
48
```

<pre>MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR <type> BUF(*)</type></pre>	1 2 3
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR) INTEGER COMM, AMODE, INFO, FH, IERROR CHARACTER*(*) FILENAME	4 5 6 7
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR) INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE	8 9 10
<pre>MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>	11 12 13 14
<pre>MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>	15 16 17
<pre>MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR <type> BUF(*)</type></pre>	18 19 20 21
<pre>MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>	22 23 24
<pre>MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>	25 26 27 28 29
<pre>MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>	30 31 32 33
<pre>MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET <type> BUF(*)</type></pre>	34 35 36 37 38
<pre>MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)     INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR     <type> BUF(*)</type></pre>	39 40 41 42
<pre>MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR <type> BUF(*)</type></pre>	42 43 44 45
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) INTEGER FH, COUNT, DATATYPE, IERROR	46 47 48

```
1
         <type> BUF(*)
\mathbf{2}
     MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
3
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
4
         <type> BUF(*)
5
6
     MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
\overline{7}
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
8
         <type> BUF(*)
9
     MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
10
         INTEGER FH, WHENCE, IERROR
11
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
12
13
     MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
14
         INTEGER FH, WHENCE, IERROR
15
         INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
16
     MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
17
         INTEGER FH, IERROR
18
         LOGICAL FLAG
19
20
     MPI_FILE_SET_INFO(FH, INFO, IERROR)
21
         INTEGER FH, INFO, IERROR
22
     MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
23
         INTEGER FH, IERROR
^{24}
         INTEGER(KIND=MPI_OFFSET_KIND) SIZE
25
26
     MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
27
         INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
28
         INTEGER(KIND=MPI_OFFSET_KIND) DISP
29
         CHARACTER*(*) DATAREP
30
     MPI_FILE_SYNC(FH, IERROR)
^{31}
         INTEGER FH, IERROR
32
33
     MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
34
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
35
         <type> BUF(*)
36
     MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
37
         INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
38
         <type> BUF(*)
39
40
     MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
41
         INTEGER FH, COUNT, DATATYPE, IERROR
42
         <type> BUF(*)
43
     MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
44
         INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
45
46
         <type> BUF(*)
47
     MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
48
```

1 INTEGER FH, COUNT, DATATYPE, STATUS(MPI\_STATUS\_SIZE), IERROR 2 INTEGER(KIND=MPI\_OFFSET\_KIND) OFFSET 3 <type> BUF(\*) 4 MPI\_FILE\_WRITE\_AT\_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR) 5INTEGER FH, COUNT, DATATYPE, STATUS(MPI\_STATUS\_SIZE), IERROR 6 INTEGER(KIND=MPI OFFSET KIND) OFFSET 7 <type> BUF(\*) 8 MPI\_FILE\_WRITE\_AT\_ALL\_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR) 9 10 INTEGER FH, COUNT, DATATYPE, IERROR 11 INTEGER(KIND=MPI\_OFFSET\_KIND) OFFSET <type> BUF(\*) 1213 MPI\_FILE\_WRITE\_AT\_ALL\_END(FH, BUF, STATUS, IERROR) 14INTEGER FH, STATUS(MPI\_STATUS\_SIZE), IERROR 15<type> BUF(\*) 1617MPI\_FILE\_WRITE\_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 18 INTEGER FH, COUNT, DATATYPE, STATUS(MPI\_STATUS\_SIZE), IERROR 19 <type> BUF(\*) 20MPI\_FILE\_WRITE\_ORDERED\_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR) 21INTEGER FH, COUNT, DATATYPE, IERROR 22 <type> BUF(\*) 2324MPI\_FILE\_WRITE\_ORDERED\_END(FH, BUF, STATUS, IERROR) 25INTEGER FH, STATUS(MPI\_STATUS\_SIZE), IERROR 26<type> BUF(\*) 27MPI\_FILE\_WRITE\_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR) 28 INTEGER FH, COUNT, DATATYPE, STATUS(MPI\_STATUS\_SIZE), IERROR 29 <type> BUF(\*) 30 31MPI\_REGISTER\_DATAREP(DATAREP, READ\_CONVERSION\_FN, WRITE\_CONVERSION\_FN, 32 DTYPE\_FILE\_EXTENT\_FN, EXTRA\_STATE, IERROR) 33 CHARACTER\*(\*) DATAREP 34 EXTERNAL READ\_CONVERSION\_FN, WRITE\_CONVERSION\_FN, DTYPE\_FILE\_EXTENT\_FN 35 INTEGER(KIND=MPI\_ADDRESS\_KIND) EXTRA\_STATE 36 INTEGER IERROR 37 38 A.5.13 Language Bindings Fortran Bindings 39 40 MPI\_F\_SYNC\_REG(BUF) 41 <type> BUF(\*) 42The following procedure is not available with mpif.h: 43 MPI\_STATUS\_F082F(F08\_STATUS, F\_STATUS, IERROR) 44TYPE(MPI\_Status) :: F08\_STATUS 45INTEGER :: F\_STATUS(MPI\_STATUS\_SIZE), IERROR 464748

```
1
     The following procedure is not available with mpif.h:
\mathbf{2}
     MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
3
         INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
4
         TYPE(MPI_Status) :: F08_STATUS
5
     MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
6
         INTEGER P, R, NEWTYPE, IERROR
7
8
     MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
9
         INTEGER R, NEWTYPE, IERROR
10
     MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
11
         INTEGER P, R, NEWTYPE, IERROR
12
13
     MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
14
         INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
15
16
     A.5.14 Tools / Profiling Interface Fortran Bindings
17
18
     MPI_PCONTROL(LEVEL)
19
         INTEGER LEVEL
20
21
22
     A.5.15 Deprecated Fortran Bindings
23
     MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
24
         INTEGER COMM, KEYVAL, IERROR
25
26
     MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
27
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
28
         LOGICAL FLAG
29
     MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
30
         INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
^{31}
32
     MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
33
                   ATTRIBUTE_VAL_OUT, FLAG, IERR)
34
         INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
35
                    ATTRIBUTE_VAL_OUT, IERR
36
         LOGICAL FLAG
37
     MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
38
         INTEGER INFO, VALUELEN, IERROR
39
         CHARACTER*(*) KEY, VALUE
40
         LOGICAL FLAG
41
42
     MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
43
         INTEGER INFO, VALUELEN, IERROR
44
         CHARACTER*(*) KEY
45
         LOGICAL FLAG
46
     MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
47
         EXTERNAL COPY_FN, DELETE_FN
48
```

A.5. FORTRAN BINDINGS WITH MPIF.H OR THE MPI MODULE	1043
INTEGER KEYVAL, EXTRA_STATE, IERROR	1
	2
MPI_KEYVAL_FREE(KEYVAL, IERROR)	3
INTEGER KEYVAL, IERROR	4
MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	5
ATTRIBUTE_VAL_OUT, FLAG, IERR)	6
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,	7
ATTRIBUTE_VAL_OUT, IERR	8
LOGICAL FLAG	9
MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)	10
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR	11 12
MPI_SIZEOF(X, SIZE, IERROR)	13
<pre><type> X</type></pre>	14
INTEGER SIZE, IERROR	15
	16
	17
	18
	19
	20
	21
	22
	23 24
	24 25
	26
	27
	28
	29
	30
	31
	32
	33
	34
	35
	36
	37 38
	39
	40
	41
	42
	43
	44
	45

### Annex B

## Change-Log

Annex B.1 summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown. If not otherwise noted, the section and page references refer to the locations of the change or new functionality in this version of the standard. Changes in Annexes B.2–B.5 were already introduced in the corresponding sections in previous versions of this standard.

B.1	Changes from Version 3.1 to Version 4.0	24
B.1.1	L Fixes to Errata in Previous Versions of MPI	25
0.1.1		26
1.	Sections 8.6.1, 8.6.2 and 8.9 on pages 417, 422 and 445, and MPI-3.1 Sections 7.6.1,	27
	7.6.2 and 7.8 on pages 315, 318 and 329.	28
	$MPI\_NEIGHBOR\_ALLTOALL\{ V W\} \ \mathrm{and} \ MPI\_NEIGHBOR\_ALLGATHER\{ V\} \ \mathrm{for} \ \mathrm{Car-}$	29
	tesian virtual grids were clarified. An advice to implementors was added to illustrate	30
	a correct implementation for the case of $periods[d] == 1$ or .TRUE. and $dims[d] == 1$ or	31
	2 in a direction d.	32
		33
2.	Section 19.3.5 on page 844, and MPI-3.1 Section 17.2.5 on page 657 line 11.	34
	Clarified that the MPI_STATUS_F2F08 and MPI_STATUS_F082F routines and the	35
	declaration for TYPE(MPI_Status) are not supposed to appear with mpif.h.	36
3.	Sections 2.5.4, 19.3.5, and A.1.1 on pages 20, 844, and 860, and MPI-3.1 Sections	37
	2.5.4, 17.2.5, and A.1.1 on pages 15, 656, and 669.	38
	Define the C constants MPI_F_STATUS_SIZE, MPI_F_SOURCE, MPI_F_TAG, and	39
	MPI_F_ERROR.	40
		41
4.	Section 19.3.5 on page 845, and MPI-3.1 Section 17.2.5 on page 658.	42
	Added missing const to IN parameters for MPI_STATUS_F2F08 and	43
	MPI_STATUS_F082F.	44
		45
		46
		47
		48

1	B.1.2	2 Changes in MPI-4.0
2 3 4	1.	Sections 2.2, 18.2.1, and 19.1.5 on pages 11, 789, and 798. The limit for the maximum length of MPI identifiers was removed.
5 6 7 8	2.	Section 2.4, 3.4, 3.7.2, 3.7.3, 3.8.1, 3.8.2, 6.13, 14.4.5, and Annex A.2 on pages 13, 49, 62, 70, 84, 87, 276, 688, and 881. The semantic terms were updated.
9 10 11 12 13 14 15	3.	Sections 2.5.8 and 19.2 on pages 22 and 839, and throughout the entire document. New large count functions MPI_{}_c in C and through function overloading in the Fortran mpi_f08 module, (with the exception of the explicit Fortran proce- dures MPI_Op_create_c and MPI_Register_datarep_c) and the new large count call- backs MPI_User_function_c and MPI_Datarep_conversion_function_c together with the predefined function MPI_CONVERSION_FN_NULL_C were introduced to accomodate large buffers and/or datatypes.
16 17 18		Clarifications were added to the behavior of INOUT/OUT parameters that cannot represent the value to be returned for the MPI_BUFFER_DETACH and MPI_FILE_GET_TYPE_EXTENT functions.
19 20		A new error class $MPI\_ERR\_VALUE\_TOO\_LARGE$ was introduced.
21 22 23 24 25 26	4.	Sections 2.8, 9.3, 9.5, and 11.2.1 on pages 26, 458, 473, and 488. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_SELF instead of MPI_COMM_WORLD. The definition of MPI_ERRORS_ARE_FATAL was clarified to cover all connected processes, and a new error handler, MPI_ERRORS_ABORT, was created to limit the scope of aborting.
27 28 29	5.	Section 3.7 on page 60. The introduction of MPI nonblocking communication was changed to describe correctness and performance reasons for the use of nonblocking communication.
30 31 32	6.	Section 3.7.2 on page 62. Addition of MPI_ISENDRECV and MPI_ISENDRECV_REPLACE.
33 34 35 36 37	7.	Sections 3.7.3, 3.9, 6.13, 8.8, and 8.9 on pages 70, 94, 276, 437, and 445. Persistent collective communication MPI_{ALLGATHER }_INIT including persistent collective neighborhood communication MPI_NEIGHBOR_{ALLGATHER }_INIT was added to the standard.
38 39 40	8.	Sections 3.8.4 and 16.3 on pages 92 and 784. Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.
41 42 43 44	9.	Chapter 4 on page 103. A new chapter on partitioned communication with the new MPI procedures $MPI_{PARRIVED PREADY{}}$ and $MPI_{PRECV PSEND}_{INIT}$ was added.
45 46 47 48	10.	Section 7.4.2 on page 327. MPI_COMM_TYPE_HW_UNGUIDED was added as a new possible value for the split_type parameter of the MPI_COMM_SPLIT_TYPE function.

11.	Section 7.4.2 on page 327. MPI_COMM_TYPE_HW_GUIDED was added as a new possible value for the split_type parameter of the MPI_COMM_SPLIT_TYPE function, as well as a new info key "mpi_hw_resource_type". A specific value associated with this new info key is also defined: "mpi_shared_memory".	1 2 3 4 5
12.	Section 7.4.2 on page 327. The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer propagate info hints. This change may affect backward compatibility.	6 7 8 9 10
13.	Section 7.4.2 on page 327. The MPI_COMM_IDUP_WITH_INFO function was added.	11 12 13
14.	Sections 7.4.4, 12.2.7, and 14.2.8 on pages 345, 567, and 653. The definition of info hints was updated to allow applications to provide assertions regarding their usage of MPI objects and operations.	14 15 16
15.	Section 7.4.4 on page 345. The new info hints "mpi_assert_no_any_tag", "mpi_assert_no_any_source", "mpi_assert_exact_length", and "mpi_assert_allow_overtaking" were added for use with communicators.	17 18 19 20 21
16.	Sections 7.4.4, 12.2.7, and 14.2.8 on pages 345, 567, and 653. The semantics of the MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, MPI_WIN_GET_INFO, MPI_FILE_SET_INFO, and MPI_FILE_GET_INFO were clarified.	21 22 23 24 25
17.	Section 8.5 on page 392. MPI_DIMS_CREATE is now guaranteed to return MPI_SUCCESS if the number of di- mensions passed to the routine is set to 0 and the number of nodes is set to 1.	26 27 28 29
18.	Sections 9.2, 12.2.2, and 12.2.3 on pages 455, 556, and 558. Introduced alignment requirements for memory allocated through MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, and MPI_WIN_ALLOCATE_SHARED and added a new info key "mpi_minimum_memory_alignment" to specify a desired alternative minimum alignment.	30 31 32 33
19.	Sections 9.3 and 9.4 on pages 458 and 469. Clarified definition of errors to say that MPI should continue whenever possible and allow the user to recover from errors.	34 35 36 37
20.	Section 9.4 on page 469. Added text to clarify what is implied about the status of MPI and user visible buffers when MPI functions return MPI_SUCCESS or other error codes.	38 39 40
21.	Section 9.4 on page 471. The error class MPI_ERR_PROC_ABORTED has been added.	41 42 43
22.	Section 10 on page 479. Added a new function MPI_INFO_GET_STRING that takes a buffer length argument for returning info value strings. This function returns the required buffer length for the requested string and guarantees null termination for C strings where buffer size is greater than 0.	44 45 46 47 48

1 2 2	23.	Section 10 on page 479 and Section 16.3 on page 784. MPI_INFO_GET and MPI_INFO_GET_VALUELEN were deprecated.
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	24.	Chapter 11, 3.2.3, 7.2.4, 7.3.2, 7.4.2, 7.6.2, 9.1.1, 9.1.2, 9.3, 9.3.4, 9.5, 11.6, 14.2.1, 14.2.7, 14.7, 15.3.4, 19.3.4, 19.3.6, and Annex A on pages 487, 35, 315, 318, 327, 358, 451, 453, 458, 466, 473, 517, 645, 651, 719, 734, 841, 846, and 857. The Sessions Model was added to the standard. New MPI procedures are MPI_SESSION_{INIT FINALIZE}, MPI_SESSION_GET_{}, MPI_SESSION_{} ERRHANDLER, MPI_GROUP_FROM_SESSION_PSET, MPI_COMM_CREATE_FROM_GROUP, MPI_INTERCOMM_CREATE_FROM_GROUPS, and new conversion functions are MPI_SESSION_{C2F F2C}. New declarations are MPI_Session in C and TYPE(MPI_Session) together with the related overloaded operators .EQ., .NE., == and /= in the Fortran mpi_f08 and mpi modules, and the callback function prototype MPI_Session_errhandler_function. New constants are MPI_SESSION_NULL, MPI_ERR_SESSION, MPI_MAX_PSET_NAME_LEN, MPI_MAX_STRINGTAG_LEN, MPI_T_BIND_MPI_SESSION and the predefined info key "mpi_size".
18 19 20	25.	Section 11.2.1 on page 488. A new function MPI_INFO_CREATE_ENV was added.
21 22 23 24	26.	Sections 11.2.1 and 11.10.4 on pages 488 and 546. Clarified the semantic of failure and error reporting before (and during) MPI_INIT and after MPI_FINALIZE.
25 26 27 28	27.	Section 11.8.4 on page 530. Added the "mpi_initial_errhandler" reserved info key with the reserved values "mpi_errors_abort", "mpi_errors_are_fatal", and "mpi_errors_return" to the launch keys in MPI_COMM_SPAWN, MPI_COMM_SPAWN_MULTIPLE, and mpiexec.
29 30 31 32	28.	Section 12.5.3 on page 602. RMA passive target synchronization using locks can now be used portably in memory allocated via MPI_WIN_ALLOCATE_SHARED.
33 34 35 36	29.	Section 13.3 on page 640. The mpi_f08 binding incorrectly had the dummy parameter flag in the MPI F08 binding for MPI_STATUS_SET_CANCELLED marked as INTENT(OUT). It has been fixed to be INTENT(IN).
37 38 39 40 41 42 43	30.	Sections 15.3 and 15.3.8 on pages 731 and 757. A callback-driven event interface with the MPI_T_{SOURCE EVENT}_{} and MPI_T_CATEGORY_{GET GET_NUM}_EVENTS routines, the declaration types MPI_T_cb_safety, MPI_T_event_{instance registration}, MPI_T_source_order, and the call- back function prototypes MPI_T_event_{cb dropped_cb free_cb}_function, was added to the MPI tool information interface.
44 45 46 47 48	31.	Section 15.3.9 on page 778. The argument stamp (previously described as a virtual time stamp) from MPI_T_CATEGORY_CHANGED was renamed to update_number and its intended im- plementation and use was clarified.

32.	Section 15.3.10, Table 15.7, and Section 16.3 on pages 778, 779, and 784. MPI_T_ERR_INVALID_ITEM is deprecated. MPI routines should return MPI_T_ERR_INVALID_INDEX instead of MPI_T_ERR_INVALID_ITEM.	1 2 3
		4
33.	Section 16.3 on page 786. MPI_SIZEOF was deprecated.	5 6
34.	Section 19.1.5 on page 798. An exception was added for the specific Fortran names in the case of TS 29113 interface specifications in mpif.h for MPI_NEIGHBOR_ALLTOALLW_INIT, MPI_NEIGHBOR_ALLTOALLV_INIT, and MPI_NEIGHBOR_ALLGATHERV_INIT.	7 8 9 10 11
B.2	Changes from Version 3.0 to Version 3.1	12 13
B.2.1	Fixes to Errata in Previous Versions of MPI	14 15
1.	Chapters 3–19, Annex A.4 on page 920, and Example 6.21 on page 238, and MPI-3.0 Chapters 3–17, Annex A.3 on page 707, and Example 5.21 on page 187. Within the mpi_f08 Fortran support method, BIND(C) was removed from all SUBROUTINE, FUNCTION, and ABSTRACT INTERFACE definitions.	16 17 18 19 20
2.	Section 3.2.5 on page 38, and MPI-3.0 Section 3.2.5 on page 30. The three public fields MPI_SOURCE, MPI_TAG, and MPI_ERROR of the Fortran derived type TYPE(MPI_Status) must be of type INTEGER.	21 22 23
3.	Section 3.8.2 on page 87, and MPI-3.0 Section 3.8.2 on page 67. The flag arguments of the Fortran interfaces of MPI_IMPROBE were originally incorrectly defined as INTEGER (instead as LOGICAL).	24 25 26 27
4.	Section 7.4.2 on page 327, and MPI-3.0 Section 6.4.2 on page 237. In the mpi_f08 binding of MPI_COMM_IDUP, the output argument newcomm is declared as ASYNCHRONOUS.	28 29 30 31
5.	Section 7.4.4 on page 345, and MPI-3.0 Section 6.4.4 on page 248. In the mpi_f08 binding of MPI_COMM_SET_INFO, the intent of comm is IN, and the optional output argument ierror was missing.	32 33 34
6.	Section 8.6 on page 416, and MPI-3.0 Sections 7.6, on pages 314. In the case of virtual general graph topolgies (created with MPI_CART_CREATE), the use of neighborhood collective communication is restricted to adjacency matrices with the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix).	35 36 37 38 39 40
7.	Section 9.1.1 on page 451, and MPI-3.0 Section 8.1.1 on page 335. In the mpi_f08 binding of MPI_GET_LIBRARY_VERSION, a typo in the resultlen argument was corrected.	41 42 43
8.	Sections 9.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR), 12.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR), 12.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR), 12.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR),	44 45 46 47 48

105	0 ANNEX B. CHANGE-LOG
1 2 3	15.2.1 and 15.2.6 (Profiling interface), and corresponding sections in MPI-3.0. The linker name concept was substituted by defining specific procedure names.
3 4 <b>(</b> 5 6 7	9. Section 12.2.1 on page 553, and MPI-3.0 Section 11.2.2 on page 407. The "same_size" info key can be used with all window flavors, and requires that all processes in the process group of the communicator have provided this info key with the same value.
8 10 10	<ol> <li>Section 12.3.4 on page 576, and MPI-3.0 Section 11.3.4 on page 424. Origin buffer arguments to MPI_GET_ACCUMULATE are ignored when the MPI_NO_OP operation is used.</li> </ol>
13 14	<ol> <li>Section 12.3.4 on page 576, and MPI-3.0 Section 11.3.4 on page 424. Clarify the roles of origin, result, and target communication parameters in MPI_GET_ACCUMULATE.</li> </ol>
15 16 12 17 18	2. Section 15.3 on page 731, and MPI-3.0 Section 14.3 on page 561 New paragraph and advice to users clarifying intent of variable names in the tools information interface.
19 20 21	3. Section 15.3.3 on page 733, and MPI-3.0 Section 14.3.3 on page 563. New paragraph clarifying variable name equivalence in the tools information interface.
22 14 23 24 25 26 27	4. Sections 15.3.6, 15.3.7, and 15.3.9 on pages 738, 744, and 773, and MPI-3.0 Sections 14.3.6, 14.3.7, and 14.3.8 on pages 567, 573, and 584. In functions MPI_T_CVAR_GET_INFO, MPI_T_PVAR_GET_INFO, and MPI_T_CATEGORY_GET_INFO, clarification of parameters that must be identical for equivalent control variable / performance variable / category names across connected processes.
28 29 15 30	5. Section 15.3.7 on page 744, and MPI-3.0 Section 14.3.7 on page 573. Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.
<sup>31</sup> 16 <sup>32</sup> <sup>33</sup>	5. Section 15.3.7 on page 744, and MPI-3.0 Section 14.3.7 on page 579, line 7. Clarify the return code when bad handle is passed to an MPI_T_PVAR_* routine.
35 36	7. Section 19.1.4 on page 797, and MPI-3.0 Section 17.1.4 on page 603. The advice to implementors at the end of the section was rewritten and moved into the following section.
37 38 18 39 40	8. Section 19.1.5 on page 798, and MPI-3.0 Section 17.1.5 on page 605. The section was fully rewritten. The linker name concept was substituted by defining specific procedure names.
41 19 42 43	9. Section 19.1.6 on page 803, and MPI-3.0 Section 17.1.6 on page 611. The requirements on BIND(C) procedure interfaces were removed.
	D. Annexes A.3, A.4, and A.5 on pages 882, 920, and 1008, and MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, and 756. The predefined callback MPI_CONVERSION_FN_NULL was added to all three an- nexes.

21. Annex A.4.5 on page 961, and MPI-3.0 Annex A.3.4 on page 724. In the mpi_f08 binding of	1 2
MPI_{COMM TYPE WIN}_{DUP NULL_COPY NULL_DELETE}_FN, all INTENT()	$\frac{3}{4}$
information was removed.	5
B.2.2 Changes in MPI-3.1	6
	7
1. Sections 2.6.4 and 5.1.5 on pages 26 and 141.	8
The use of the intrinsic operators "+" and "-" for absolute addresses is substituted by MPI_AINT_ADD and MPI_AINT_DIFF. In C, they can be implemented as macros.	9 10
2. Sections 9.1.1, 11.2.1, and 11.6 on pages 451, 488, and 517.	11
The routines MPI_INITIALIZED, MPI_FINALIZED, MPI_QUERY_THREAD,	12
MPI_IS_THREAD_MAIN, MPI_GET_VERSION, and MPI_GET_LIBRARY_VERSION	13 14
are callable from threads without restriction (in the sense of $MPI\_THREAD\_MULTIPLE),$	14
irrespective of the actual level of thread support provided, in the case where the im-	16
plementation supports threads.	17
3. Section 12.2.1 on page 553.	18
The "same_disp_unit" info key was added for use in RMA window creation routines.	19
4. Sections 14.4.2 and 14.4.3 on pages 662 and 669.	20
Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL,	21
MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL	22 23
	23 24
5. Sections 15.3.6, 15.3.7, and 15.3.9 on pages 738, 744, and 773.	25
Clarified that NULL parameters can be provided in	26
MPI_T_{CVAR PVAR CATEGORY}_GET_INFO routines.	27
6. Sections 15.3.6, 15.3.7, 15.3.9, and 15.3.10 on pages 738, 744, 773, and 778.	28
New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVAR_GET_INDEX,	29
MPI_T_CATEGORY_GET_INDEX, were added to support retrieving indices of vari-	30
ables and categories. The error codes $MPI\_T\_ERR\_INVALID$ and	31
MPI_T_ERR_INVALID_NAME were added to indicate invalid uses of the interface.	32
	33
B.3 Changes from Version 2.2 to Version 3.0	34 35
	36
B.3.1 Fixes to Errata in Previous Versions of MPI	37
1. Sections 2.6.2 and 2.6.3 on pages 24 and 25, and MPI-2.2 Section 2.6.2 on page 17,	38
lines $41-42$ , Section 2.6.3 on page 18, lines $15-16$ , and Section 2.6.4 on page 18,	39
lines 40–41.	40
This is an $MPI-2$ erratum: The scope for the reserved prefix $\mathtt{MPI}\_$ and the C++	41
namespace MPI is now any name as originally intended in MPI-1.	42
2. Sections 3.2.2, 6.9.2, 14.5.2 Table 14.2, and Annex A.1.1 on pages 33, 226, 702, and	43
857, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and	44
Annex A.1.1 on pages 27, 164, 433, 472 and 513	45
This is an MPI-2.2 erratum: New named predefined datatypes	46
MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and	47 48
	10

1052	ANNEX B. CHANGE-LOG
	MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding to the C++ types bool, std::complex <float>, std::complex<double>, and std::complex<long double="">. These datatypes also correspond to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The nonstandard C++ types Complex&lt;&gt; were substituted by the standard types std::complex&lt;&gt;</long></double></float>
3.	Sections 6.9.2 on pages 226 and MPI-2.2 Section 5.9.2, page 165, line 47. This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" reduction group.
4.	Section 8.5.5 on page 403, and MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3.

This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are now defined as int& indegree and int& outdegree in the C++ interface of MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT.

- 5. Section 14.5.2, Table 14.2 on page 702, and MPI-2.2, Section 13.5.3, Table 13.2 on page 433.
  - This was an MPI-2.2 erratum: The MPI\_C\_BOOL "external32" representation is corrected to a 1-byte size.
  - 6. MPI-2.2 Section 16.1.16 on page 471, line 45. This is an MPI-2.2 erratum: The constant MPI::\_LONG\_LONG should be MPI::LONG\_LONG.
  - 7. Annex A.1.1 on page 857, Table "Optional datatypes (Fortran)," and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37–41. This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F\_COMPLEX4, MPI::F\_COMPLEX8, MPI::F\_COMPLEX16, MPI::F\_COMPLEX32 were added to the table.

#### B.3.2 Changes in MPI-3.0

- 1. Section 2.6.1 on page 23, Section 17.2 on page 788 and all other chapters. The C++ bindings were removed from the standard. See errata in Section B.3.1 on page 1051 for the latest changes to the MPI C++ binding defined in MPI-2.2. This change may affect backward compatibility.
- 2. Section 2.6.1 on page 23, Section 16.1 on page 781 and Section 17.1 on page 787. The deprecated functions MPI\_TYPE\_HVECTOR, MPI\_TYPE\_HINDEXED, MPI\_TYPE\_STRUCT, MPI\_ADDRESS, MPI\_TYPE\_EXTENT, MPI\_TYPE\_LB, MPI\_TYPE\_UB, MPI\_ERRHANDLER\_CREATE (and its callback function prototype MPI\_Handler\_function), MPI\_ERRHANDLER\_SET, MPI\_ERRHANDLER\_GET, the dep-recated special datatype handles MPI\_LB, MPI\_UB, and the constants MPI\_COMBINER\_HINDEXED\_INTEGER, MPI\_COMBINER\_HVECTOR\_INTEGER, MPI\_COMBINER\_STRUCT\_INTEGER were removed from the standard. This change may affect backward compatibility.

3.	Section 2.3 on page 12. Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.	1 2 3
4.	Section 2.5.4 on page 20 and Section 8.5.4 on page 396. The recommended C implementation value for MPI_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was in- troduced.	4 5 6 7 8
5.	Section 2.5.4 on page 20 and Section 9.1.1 on page 451. Added the new routine MPI_GET_LIBRARY_VERSION to query library specific versions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.	9 10 11 12
6.	Sections 2.5.8, 3.2.2, 3.3, 6.9.2, on pages 22, 33, 35, 226, Sections 5.1, 5.1.7, 5.1.8, 5.1.11, 13.3 on pages 119, 147, 149, 153, 640, and Annex A.1.1 on page 857. New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their results as an MPI_Count value, which is a new type large enough to represent element counts in memory, file views, etc. A new function, MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object so that a call to MPI_GET_ELEMENTS_X returns the provided MPI_Count value (in Fortran, INTEGER(KIND=MPI_COUNT_KIND)). The corresponding predefined datatype is MPI_COUNT.	13 14 15 16 17 18 19 20 21 22
7.	Chapter 3 on page 31 through Chapter 19 on page 791. In the C language bindings, the array-arguments' interfaces were modified to consistently use use [] instead of *. Exceptions are MPI_INIT, which continues to use char ***argv (correct because of subtle rules regarding the use of the & operator with char *argv[]), and	23 24 25 26 27 28
8.	MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT. Sections 3.2.5, 5.1.5, 5.1.11, 5.2 on pages 38, 141, 153, 174. The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the count argument to MPI_UNDEFINED when that argument would overflow. The functions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 routines, the type and semantics of the count arguments remain unchanged, i.e., int or INTEGER.	29 30 31 32 33 34 35 36 37
9.	Section 3.2.6 on page 41, and Section 3.8 on page 84. MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE.	38 39 40 41
10.	Section 3.8 on page 84 and Section 3.10 on page 101. The use of MPI_PROC_NULL in probe operations was clarified. A special predefined message MPI_MESSAGE_NO_PROC was defined for the use of matching probe (i.e., the new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.	42 43 44 45
11.	Sections 3.8.2, 3.8.3, 19.3.4, A.1.1 on pages 87, 90, 841, 857. Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and MPI_IMPROBE	46 47 48

1 2 3 4 5 6		operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message with the new routines MPI_MRECV and MPI_IMRECV regardless of other intervening probe or receive operations. The opaque object MPI_Message, the null handle MPI_MESSAGE_NULL, and the conversion functions MPI_Message_c2f and MPI_Message_f2c were defined.
7 8 9 10	12.	Section 5.1.2 on page 121 and Section 5.1.13 on page 157. The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK were added.
11 12 13	13.	Chapter 6 on page 187 and Section 6.12 on page 250. Added nonblocking interfaces to all collective operations.
14 15 16 17	14.	Sections 7.4.2, 7.4.4, 12.2.7, on pages 327, 345, 567. The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.
18 19 20	15.	Section 7.4.2 on page 327. Added MPI_COMM_IDUP.
21 22 23 24 25	16.	Section 7.4.2 on page 327. Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.
26 27 28	17.	Section 7.4.2 on page 327. Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type con- stant MPI_COMM_TYPE_SHARED.
29 30 31 32 33	18.	Section 7.6.2 on page 358. In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation could interfere with point-to-point communication on the parent communicator with the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0.
34 35 36 37	19.	Section 7.8 on page 381. Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.
38 39	20.	Section 8.5.8 on page 414. MPI_CART_MAP can also be used for a zero-dimensional topologies.
40 41 42 43 44 45 46 47 48	21.	Section 8.6 on page 416 and Section 8.7 on page 429. The following neighborhood collective communication routines were added to support sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in

MPI\_NEIGHBOR\_ALLTOALLW and MPI\_INEIGHBOR\_ALLTOALLW were defined as address size integers. In MPI\_DIST\_GRAPH\_NEIGHBORS, an ordering rule was added for communicators created with MPI\_DIST\_GRAPH\_CREATE\_ADJACENT.

Section 11.2.1 on page 488 and Section 11.2.1 on page 491.	4 5
The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After MPI is initialized, the application can access information about the execution environment by querying the new predefined info object MPI_INFO_ENV.	6 7 8
Section 11.2.1 on page 488. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.	9 10 11
Chapter 12 on page 551. Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target communication, new one-sided communication routines, a new memory model, and other changes.	12 13 14 15
Section 15.3 on page 731. A new MPI Tool Information Interface was added.	16 17 18
The following changes are related to the Fortran language support.	19
Section 2.3 on page 12, and Sections 19.1.1, 19.1.2, 19.1.7 on pages 791, 792, and 807. The new mpi_08 Fortran module was introduced.	20 21 22
Section 2.5.1 on page 18, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 792, 795, and 807. Handles to opaque objects were defined as named types within the mpi_08 Fortran module. The operators .EQ., .NE., ==, and /= were overloaded to allow the comparison of these handles. The handle types and the overloaded operators are also available through the mpi Fortran module.	23 24 25 26 27 28
Sections 2.5.4, 2.5.5 on pages 20, 21, Sections 19.1.1, 19.1.10, 19.1.11, 19.1.12, 19.1.13 on pages 791, 817, 819, 819, 822, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 792, 795, 807.	29 30 31
Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [46], and the compile-time constant MPI_SUBARRAYS_SUPPORTED was set to .TRUE With this, Fortran subscript triplets can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TS 29113 feature, the constant is set to .FALSE	32 33 34 35 36 37
Section 2.6.2 on page 24, Section 19.1.2 on page 792, and Section 19.1.7 on page 807. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.	38 39 40
<ul> <li>Section 3.2.5 on page 38, Sections 19.1.2, 19.1.3, 19.1.7, on pages 792, 795, 807, and Section 19.3.5 on page 843.</li> <li>Within the mpi_08 Fortran module, the status was defined as TYPE(MPI_Status). Additionally, within both the mpi and the mpi_f08 modules, the constants</li> <li>MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined. New conversion routines were added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, In mpi.h, the new</li> </ul>	41 42 43 44 45 46 47 48
	The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After MPI is initialized, the application can access information about the execution envi- ronment by querying the new predefined info object MPI_INFO_ENV. Section 11.2.1 on page 488. Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE. Chapter 12 on page 551. Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target communication, new one-sided communication routines, a new memory model, and other changes. Section 15.3 on page 731. A new MPI Tool Information Interface was added. The following changes are related to the Fortran language support. Section 2.3 on page 12, and Sections 19.1.1, 19.1.2, 19.1.7 on pages 791, 792, and 807. The new mpi_08 Fortran module was introduced. Section 2.5.1 on page 18, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 792, 795, and 807. Handles to opaque objects were defined as named types within the mpi_08 Fortran module. The operators .EQ., .NE., ==, and /= were overloaded to allow the compari- son of these handles. The handle types and the overloaded operators are also available through the mpi Fortran module. Sections 2.5.4, 2.5.5 on pages 20, 21, Sections 19.1.1, 19.1.0, 19.1.11, 19.1.12, 19.1.13 on pages 791, 817, 819, 819, 822, and Sections 19.1.2, 19.1.3, 19.1.7 on pages 792, 795, 807. Within the mpi_08 Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [46], and the compile-time constant MPI_SUBARRAYS_SUPPORTED was set to .TRUE With this, Fortran Subscript triplets can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TS 29113 feature, the constant is set to .FALSE Section 3.2.5 on page 38, Sections 19.1.2, 19.1.3, 19.1.7, on pages 792, 795, 807, and Section 19.3.5 on page 843. Within the mpi_08 Fortran module, the status was defined as TYPE(M

1

 $^{2}$ 

2

3

4

5

6 7

8

9

10

11

12

13

14

30

 $^{31}$ 

32

33

34 35

36

37

38

39 40

41

42

43

44

45

type MPI\_F08\_status, and the external variables MPI\_F08\_STATUS\_IGNORE and MPI\_F08\_STATUSES\_IGNORE were added.

31. Section 3.6 on page 57.

In Fortran with the mpi module or mpif.h, the type of the buffer\_addr argument of MPI\_BUFFER\_DETACH is incorrectly defined and the argument is therefore unused.

32. Section 5.1 on page 119, Section 5.1.6 on page 144, and Section 19.1.15 on page 823. The Fortran alignments of basic datatypes within Fortran derived types are implementation dependent; therefore it is recommended to use the BIND(C) attribute for derived types in MPI communication buffers. If an array of structures (in C/C++) or derived types (in Fortran) is to be used in MPI communication buffers, it is recommended that the user creates a portable datatype handle and additionally applies MPI\_TYPE\_CREATE\_RESIZED to this datatype handle.

- <sup>15</sup> 33. Sections 5.1.10, 6.9.5, 6.9.7, 7.7.4, 7.8, 9.3.1, 9.3.2, 9.3.3, 16.1, 19.1.9 on pages 152,
  <sup>16</sup> 233, 240, 376, 381, 461, 463, 465, 781, and 809. In some routines, the dummy argument names were changed because they were identical to the Fortran keywords
  <sup>18</sup> TYPE and FUNCTION. The new dummy argument names must be used because the
  <sup>19</sup> mpi and mpi\_08 modules guarantee keyword-based actual argument lists. The argument name type was changed in MPI\_TYPE\_DUP, the Fortran
- <sup>21</sup> USER\_FUNCTION of MPI\_OP\_CREATE, MPI\_TYPE\_SET\_ATTR,
- <sup>22</sup> MPI\_TYPE\_GET\_ATTR, MPI\_TYPE\_DELETE\_ATTR, MPI\_TYPE\_SET\_NAME,
- <sup>23</sup> MPI\_TYPE\_GET\_NAME, MPI\_TYPE\_MATCH\_SIZE, the callback prototype defini-
- <sup>24</sup> tion MPI\_Type\_delete\_attr\_function, and the predefined callback function
- <sup>25</sup> MPI\_TYPE\_NULL\_DELETE\_FN; function was changed in MPI\_OP\_CREATE,
- <sup>26</sup> MPI\_COMM\_CREATE\_ERRHANDLER, MPI\_WIN\_CREATE\_ERRHANDLER,
- MPI\_FILE\_CREATE\_ERRHANDLER, and MPI\_ERRHANDLER\_CREATE. For consistency reasons, INOUBUF was changed to INOUTBUF in MPI\_REDUCE\_LOCAL, and intracomm to newintracomm in MPI\_INTERCOMM\_MERGE.
  - 34. Section 7.7.2 on page 366.
    - It was clarified that in Fortran, the flag values returned by a comm\_copy\_attr\_fn callback, including MPI\_COMM\_NULL\_COPY\_FN and MPI\_COMM\_DUP\_FN, are .FALSE. and .TRUE.; see MPI\_COMM\_CREATE\_KEYVAL.

35. Section 9.2 on page 455.With the mpi and mpi\_f08 Fortran modules, MPI\_ALLOC\_MEM now also supports TYPE(C\_PTR) C-pointers instead of only returning an address-sized integer that may be usable together with a nonstandard Cray-pointer.

- 36. Section 19.1.15 on page 823, and Section 19.1.7 on page 807.Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.
- 37. Section 19.1.16 on page 825 to Section 19.1.19 on page 834, Section 19.1.7 on page 807, and Section 19.1.8 on page 808.
- The sections about Fortran optimization problems and their solutions were partially rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute. The constant MPI\_ASYNC\_PROTECTS\_NONBLOCKING tells whether the semantics of

the ASYNCHRONOUS attribute is extended to protect nonblocking operations. The Fortran routine MPI\_F\_SYNC\_REG is added. MPI-3.0 compliance for an MPI library together with a Fortran compiler is defined in Section 19.1.7.

38. Section 19.1.2 on page 792. 5Within the mpi\_08 Fortran module, dummy arguments are now declared with 6 INTENT=IN, OUT, or INOUT as defined in the mpi\_08 interfaces. 7 39. Section 19.1.3 on page 795, and Section 19.1.7 on page 807. 9 The existing mpi Fortran module must implement compile-time argument checking. 10 11 40. Section 19.1.4 on page 797. 12The use of the mpif.h Fortran include file is now strongly discouraged. 13 41. Section A.1.1, Table "Predefined functions" on page 865, Section A.1.3 on page 872, 14and Section A.4.5 on page 961. 15Within the new mpi\_f08 module, all callback prototype definitions are now defined 16 with explicit interfaces PROCEDURE(MPI\_...) that have the BIND(C) attribute; user-17 written callbacks must be modified if the mpi\_f08 module is used. 18 1942. Section A.1.3 on page 872. 20In some routines, the Fortran callback prototype names were changed from ...\_FN to 21...\_FUNCTION to be consistent with the other language bindings. 22 23Changes from Version 2.1 to Version 2.2 B.4  $^{24}$ 251. Section 2.5.4 on page 20. 26It is now guaranteed that predefined named constant handles (as other constants) 27can be used in initialization expressions or assignments, i.e., also before the call to 28 MPI\_INIT. 29 30 2. Section 2.6 on page 23, and Section 17.2 on page 788. 31The C++ language bindings have been deprecated and may be removed in a future 32 version of the MPI specification. 33 3. Section 3.2.2 on page 33. 34 MPI\_CHAR for printable characters is now defined for C type char (instead of signed 35char). This change should not have any impact on applications nor on MPI libraries 36 (except some comment lines), because printable characters could and can be stored in 37 any of the C types char, signed char, and unsigned char, and MPI\_CHAR is not allowed 38 for predefined reduction operations. 39 40 4. Section 3.2.2 on page 33. 41 MPI\_(U)INT{8,16,32,64}\_T, MPI\_AINT, MPI\_OFFSET, MPI\_C\_BOOL, 42MPI\_C\_COMPLEX, MPI\_C\_FLOAT\_COMPLEX, MPI\_C\_DOUBLE\_COMPLEX, and 43 MPI\_C\_LONG\_DOUBLE\_COMPLEX are now valid predefined MPI datatypes. 44455. Section 3.4 on page 49, Section 3.7.2 on page 62, Section 3.9 on page 94, and Section 6.1 46on page 187. 47The read access restriction on the send buffer for blocking, non blocking and collective

1

2

3

4

1 2 2		API has been lifted. It is permitted to access for read the send buffer while the operation is in progress.
3 4 5	6.	Section 3.7 on page 60. The Advice to users for IBSEND and IRSEND was slightly changed.
6 7 8 9	7.	Section 3.7.3 on page 70. The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE.
10 11 12	8.	Section 3.7.6 on page 83. MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.
12 13 14 15	9.	Section 6.8 on page 217. "In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and MPI_ALLTOALLW for intra-communicators.
16 17 18 19 20	10.	Section 6.9.2 on page 226. Predefined parameterized datatypes (e.g., returned by MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g. MPI_REAL8) have been added to the list of valid datatypes in reduction operations.
21 22 23 24 25 26	11.	Section 6.9.2 on page 226. MPI_(U)INT{8,16,32,64}_T are all considered C integer types for the purposes of the predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran integer types. MPI_C_BOOL is considered a Logical type. MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types.
27 28 29 30	12.	Section 6.9.7 on page 240. The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added.
31 32 33 34	13.	Section 6.10.1 on page 242. The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI stan- dard.
35 36	14.	Section 6.11.2 on page 247. Added in place argument to MPI_EXSCAN.
37 38 39 40 41 42 43	15.	Section 7.4.2 on page 327, and Section 7.6 on page 355. Implementations that did not implement MPI_COMM_CREATE on inter-communi- cators will need to add that functionality. As the standard described the behav- ior of this operation on inter-communicators, it is believed that most implementa- tions already provide this functionality. Note also that the C++ binding for both MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms.
44 45 46 47 48	16.	Section 7.4.2 on page 327. MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intra-communicator. If comm is an inter-communicator it was clarified that all processes in the same local group of comm must specify the same value for group.

17.	Section 8.5.4 on page 396. New functions for a scalable distributed graph topology interface has been added. In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++ class Distgraphcomm were added.	1 2 3 4 5
18.	Section 8.5.5 on page 403. For the scalable distributed graph topology interface, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS and the constant MPI_DIST_GRAPH were added.	6 7 8 9 10
19.	Section 8.5.5 on page 403. Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.	11 12 13 14
20.	Section 9.1.1 on page 451. The subversion number changed from 1 to 2.	14 15 16
21.	Section 9.3 on page 458, Section 16.2 on page 784, and Annex A.1.3 on page 872. Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.	17 18 19 20
22.	Section 11.2.4 on page 498. Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Implementors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.	21 22 23 24
23.	Section 12.3.4 on page 576. The restriction added in MPI 2.1 that the operation MPI_REPLACE in MPI_ACCUMULATE can be used only with predefined datatypes has been removed. MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others.	25 26 27 28 29 30 31 32
24.	Section 13.2 on page 633. Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments.	33 34 35 36
25.	Section 14.5.2 on page 701, and Table 14.2 on page 702. MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes in the "external32" representation.	37 38 39 40 41 42
26.	Section 19.3.7 on page 849. The description was modified that it only describes how an MPI implementation behaves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 16.17 was replaced with three new examples 19.13, 19.14, and 19.15 on pages 849–851 explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated.	43 44 45 46 47 48

	1060	ANNEX B. CHANGE-LOG
1 2	27.	Annex A.1.1 on page 857. Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 871).
3 4 5 6 7 8	28.	Annex A.1.1 on page 857. Table Named Predefined Datatypes. Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.
9 10	B.5	Changes from Version 2.0 to Version 2.1
11 12 13	1.	Section 3.2.2 on page 33, and Annex A.1 on page 857. In addition, the MPI_LONG_LONG should be added as an optional type; it is a synonym for MPI_LONG_LONG_INT.
14 15 16 17 18	2.	Section 3.2.2 on page 33, and Annex A.1 on page 857. MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG_LONG, MPI_SIGNED_CHAR, and MPI_WCHAR are moved from optional to official and they are therefore defined for all three language bindings.
19 20 21 22 23 24	3.	Section 3.2.5 on page 38. MPI_GET_COUNT with zero-length datatypes: The value returned as the count argument of MPI_GET_COUNT for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, MPI_UNDEFINED is returned.
25 26 27 28 29	4.	Section 5.1 on page 119. General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.
30 31 32 33	5.	Section 5.3 on page 182. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.
34 35 36 37 38	6.	Section 6.9.6 on page 238. If comm is an inter-communicator in MPI_ALLREDUCE, then both groups should provide count and datatype arguments that specify the same type signature (i.e., it is not necessary that both groups provide the same count value).
39 40 41 42	7.	Section 7.3.1 on page 316. MPI_GROUP_TRANSLATE_RANKS and MPI_PROC_NULL: MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.
43 44 45	8.	Section 7.7 on page 365. About the attribute caching functions:
46 47 48		Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to

	MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each key- val, information on the type of the associated user function. ( <i>End of advice to</i> <i>implementors.</i> )	1 2 3 4
9.	Section 7.8 on page 381. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In For- tran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT_NAME.	5 6 7 8 9 10
10.	Section 8.4 on page 391. About MPI_GRAPH_CREATE and MPI_CART_CREATE: All input arguments must have identical values on all processes of the group of comm_old.	11 12 13 14
11.	Section 8.5.1 on page 392. In MPI_CART_CREATE: If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.	15 16 17 18
12.	Section 8.5.3 on page 394. In MPI_GRAPH_CREATE: If the graph is empty, i.e., nnodes $== 0$ , then MPI_COMM_NULL is returned in all processes.	19 20 21 22
13.	Section 8.5.3 on page 394. In MPI_GRAPH_CREATE: A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be nonsymmetric.	23 24 25 26 27
	Advice to users. Performance implications of using multiple edges or a nonsymmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. ( <i>End of advice to users.</i> )	28 29 30 31
14.	Section 8.5.5 on page 403. In MPI_CARTDIM_GET and MPI_CART_GET: If comm is associated with a zero- dimensional Cartesian topology, MPI_CARTDIM_GET returns ndims=0 and MPI_CART_GET will keep all output arguments unchanged.	32 33 34 35 36
15.	Section 8.5.5 on page 403. In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topol- ogy, coord is not significant and 0 is returned in rank.	37 38 39
16.	Section 8.5.5 on page 403. In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.	40 41 42 43
17.	Section 8.5.6 on page 412. In MPI_CART_SHIFT: It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.	44 45 46 47 48

1 2 3 4 5	18.	Section 8.5.7 on page 413. In MPI_CART_SUB: If all entries in remain_dims are false or comm is already associated with a zero-dimensional Cartesian topology then newcomm is associated with a zero-dimensional Cartesian topology.
6 7	18.1.	Section $9.1.1$ on page $451$ . The subversion number changed from 0 to 1.
8 9 10 11 12 13	19.	Section 9.1.2 on page 453. In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.
14 15 16 17 18 19 20	20.	Section 9.3 on page 458. MPI_{COMM,WIN,FILE}_GET_ERRHANDLER behave as if a new error handler object is created. That is, once the error handler is no longer needed, MPI_ERRHANDLER_FREE should be called with the error handler returned from MPI_ERRHANDLER_GET or MPI_{COMM,WIN,FILE}_GET_ERRHANDLER to mark the error handler for deallocation. This provides behavior similar to that of MPI_COMM_GROUP and MPI_GROUP_FREE.
21 22 23 24 25 26	21.	Section 11.2.1 on page 488, see explanations to MPI_FINALIZE. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 11.10.4 on page 546.
27 28 29	22.	Section 11.2.1 on page 488. About MPI_ABORT:
30 31 32 33		Advice to users. Whether the errorcode is returned from the executable or from the MPI process startup mechanism (e.g., mpiexec), is an aspect of quality of the MPI library but not mandatory. ( <i>End of advice to users.</i> )
34 35 36 37		Advice to implementors. Where possible, a high-quality implementation will try to return the errorcode from the MPI process startup mechanism (e.g. mpiexec or singleton init). (End of advice to implementors.)
<ol> <li>37</li> <li>38</li> <li>39</li> <li>40</li> <li>41</li> <li>42</li> <li>43</li> <li>44</li> <li>45</li> </ol>	23.	Section 10 on page 479. An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET_must retain all (key,value) pairs so that layered functionality can also use the Info object.
46 47 48	24.	Section 12.3 on page 569. MPI_PROC_NULL is a valid target rank in the MPI RMA calls MPI_ACCUMULATE,

	MPI_GET, and MPI_PUT. The effect is the same as for MPI_PROC_NULL in MPI point-to-point communication. See also item 25 in this list.	1 2
25.	Section 12.3 on page 569. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the	3 4 5
	RMA epoch with the synchronization method that started the epoch. See also item 24 in this list.	6 7
26.	Section 12.3.4 on page 576.	8
	MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.	9 10 11
27.	Section 14.2.8 on page 653.	12
	About MPI_FILE_SET_VIEW and MPI_FILE_SET_INFO: When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or	13 14
	MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.	15 16 17
28.	Section 14.2.8 on page 653.	18
	About MPI_FILE_GET_INFO: If no hint exists for the file associated with fh, a handle to a newly created info object is returned that contains no key/value pair.	19 20
29.	Section $14.3$ on page $656$ .	21
	If a file does not have the mode MPI_MODE_SEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.	22 23 24
30.	Section $14.5.2$ on page 701.	25
	The bias of 16 byte doubles was defined with 10383. The correct value is 16383.	26
31.	MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0).	27 28
	In the example in this section, the buffer should be declared as const void* buf.	29
32.	Section 19.1.9 on page 809.	30
	About MPI_TYPE_CREATE_F90_XXX:	31 32
	Advice to implementors. An application may often repeat a call to	33
	$MPI\_TYPE\_CREATE\_F90\_XXX$ with the same combination of $(XXX,p,r).$ The	34
	application is not allowed to free the returned predefined, unnamed datatype	35
	handles. To prevent the creation of a potentially huge amount of handles, the	36 37
	MPI implementation should return the same datatype handle for the same ( REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (	38
	p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash-	39
	table to find formerly generated handles should limit the overhead of finding	40
	a previously generated datatype with same combination of (XXX,p,r). (End of	41
	advice to implementors.)	42
<b>9</b> 9	Section A 1.1 on page 257	43
<b>ე</b> ე.	Section A.1.1 on page 857. MPI_BOTTOM is defined as void * const MPI::BOTTOM.	44
		45
		46 47
		48

# Bibliography

[1]	Reverse domain name notation convention. https://docs.oracle.com/javase/ tutorial/java/package/namingpkgs.html. Citation on page 344.	11 12
[2]	V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and com- munication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. Citation on page 2.	13 14 15 16 17
[3]	V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. Citation on page 2.	18 19 20 21
[4]	Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In <i>OON-SKI '94</i> , page in press, 1994. Citation on page 311.	21 22 23
[5]	A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. <i>IEEE Computer</i> , 26(6):88–95, June 1993. Citation on page 2.	24 25 26 27
[6]	Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. <i>Parallel Computing</i> , 15:119–132, 1990. Citation on page 2.	28 29 30
[7]	Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. $IJHPCN$ , $1(1/2/3)$ :91–99, 2004. Citation on page 611.	31 32 33 34
[8]	Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and eval- uation of primitives for parallel I/O. In <i>Proceedings of Supercomputing '93</i> , pages 452–461, 1993. Citation on page 643.	35 36 37 38
[9]	R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992. Citation on page 2.	39 40
[10]	Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. <i>Parallel Computing</i> , 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. Citation on page 2.	41 42 43 44 45
[11]	Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. <i>Parallel Computing</i> , 20(4):615–632, April 1994. Citation on page 2.	46 47 48

1 [12] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-2 routed networks for interprocessor communication in multicomputers. In Proceedings 3 of the 1990 Supercomputing Conference, pages 647–656, 1990. Citation on page 389. 4 [13] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication 5performance of large multicomputers. In Proceedings of the 1991 International Con-6 ference on Parallel Processing, vol. II (Software), pages II-1-II-4, 1991. Citation on 7 page 389. 8 9 [14] Parasoft Corporation. Express version 1.0: A communication environment for parallel 10 computers, 1988. Citation on page 2. 1112[15] Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, 13editors. Recent Advances in the Message Passing Interface - 18th European MPI Users? 14Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings, 15volume 6960 of Lecture Notes in Computer Science. Springer, 2011. Citations on pages 161066 and 1068. 17[16] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel 18 I/O via a two-phase run-time access strategy. In IPPS '93 Workshop on Input/Output 19in Parallel Computer Systems, pages 56–70, 1993. Also published in Computer Archi-20tecture News 21(5), December 1993, pages 31–38. Citation on page 643. 2122[17] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar 23Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator cre- $^{24}$ ation in MPI. In Cotronis et al. [15], pages 282–291. Citation on page 334. 2526[18] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework 27supports heterogeneous network computing. Computers in Physics, 7(2):166–75, April 281993. Citation on page 2. 29[19] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-30 level, message passing interface in a distributed memory environment. Technical Report  $^{31}$ TM-12231, Oak Ridge National Laboratory, February 1993. Citation on page 2. 32 33 [20] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, 34June 1991. Citation on page 2. 35 36 [21] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 37 Interface, May 1992. Citation on page 2. 38 [22] D. Feitelson. Communicators: Object-based multiparty interactions for parallel pro-39 gramming. Technical Report 91-12, Dept. Computer Science, The Hebrew University 40of Jerusalem, November 1991. Citation on page 312. 41 42[23] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. The In-43 ternational Journal of Supercomputer Applications and High Performance Computing, 448, 1994. Citation on page 2. 4546 [24] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 471.1). Technical report, 1995. http://www.mpi-forum.org. Citation on page 3. 48

- [25] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing. MIT Press, 1994. Citation on page 487.
- [26] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. Citation on page 2.
- [27] Brice Goglin, Emmanuel Jeannot, Farouk Mansouri, and Guillaume Mercier. Hardware topology management in MPI applications through hierarchical communicators. *Parallel Computing*, 76:70–90, 2018. Citation on page 343.
- [28] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded MPI communication. In ISC High Performance Conference (ISC), 2019. Citation on page 103.
- [29] Ryan E. Grant, Anthony Skjellum, and Purushotham V. Bangalore. Lightweight threading with MPI using persistent communications semantics. In Workshop on Exascale MPI (ExaMPI). Held in conjunction with the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), 2015. Citation on page 103.
- [30] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. Citation on page 87.
- [31] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. Citation on page 2.
- [32] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Citation on page 796.
- [33] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. Citation on page 87.
- [34] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. Citation on page 250.
- [35] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. Citation on page 417.
- [36] T. Hoefler and A. Lumsdaine. Message progression in parallel computing to thread or not to thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. Citation on page 250.

3

4

5

6

7 8

9

10

11 12

13

14 15

16

17

18

19

20 21

22

23

 $^{24}$ 

25 26

27 28

29

30

31 32

33

34

35

36

37

38 39

40

41

42

43

 $\frac{44}{45}$ 

46

47

1 2 3 4	[37]	T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In <i>Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07.</i> IEEE Computer Society/ACM, Nov. 2007. Citation on page 252.
5 6 7 8 9	[38]	T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimiza- tion for medical image reconstruction algorithms. In <i>Recent Advances in Parallel Vir-</i> <i>tual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group</i> <i>Meeting</i> , volume LNCS 5205, pages 75–83. Springer, Sep. 2008. Citation on page 250.
10 11 12 13	[39]	T. Hoefler and J. L. Träff. Sparse collective operations for MPI. In <i>Proceedings of the 23rd IEEE International Parallel &amp; Distributed Processing Symposium, HIPS'09 Workshop</i> , May 2009. Citation on page 417.
14 15	[40]	Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI — common practice, issues, and extensions. In Cotronis et al. [15], pages 345–355. Citation on page 330.
16 17 18 19 20	[41]	Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. <i>Parallel Computing</i> , 81:32 – 57, 2019. Citation on page 276.
21 22	[42]	Institute of Electrical and Electronics Engineers, New York. <i>IEEE Standard for Binary Floating-Point Arithmetic, IEEE Standard 754-2008</i> , 2008. Citation on page 701.
23 24 25 26	[43]	International Organization for Standardization, Geneva. ISO 8859-1:1987: Informa- tion processing — 8-bit single-byte coded graphic character sets — Part 1: Latin al- phabet No. 1, February 1987. Citation on page 701.
27 28 29 30	[44]	International Organization for Standardization, Geneva. ISO/IEC 9945-1:1996: Infor- mation technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language], December 1996. Citations on pages 517 and 647.
31 32 33 34	[45]	International Organization for Standardization, Geneva. ISO/IEC 1539-1:2010: Information technology — Programming languages — Fortran — Part 1: Base language, November 2010. Citations on pages 791 and 794.
35 36 37 38	[46]	International Organization for Standardization, Geneva. <i>ISO/IEC TS 29113:2012:</i> Information technology — Further interoperability of Fortran with C, December 2012. Citations on pages 791, 792, 794, 807, 808, and 1055.
39 40 41	[47]	Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. <i>The High Performance Fortran Handbook</i> . MIT Press, 1993. Citation on page 135.
42 43 44 45 46	[48]	David Kotz. Disk-directed I/O for MIMD multiprocessors. In <i>Proceedings of the 1994 Symposium on Operating Systems Design and Implementation</i> , pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. Citation on page 643.
47 48	[49]	O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. <i>Parallel Computing</i> , 9:213–225, 1989. Citation on page 389.

- [50] S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Available online: https://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf. Citation on page 548.
- [51] Message Passing Interface Forum. Summary of the semantics of all operation-related MPI procedures, 2020. Available online: https://www.mpi-forum.org/docs. Citation on page 881.
- [52] Bradley Morgan, Daniel J. Holmes, Anthony Skjellum, Purushotham Bangalore, and Srinivas Sridharan. Planning for performance: Persistent collective operations for MPI. In Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17, pages 4:1–4:11, New York, NY, USA, 2017. ACM. Citation on page 276.
- [53] nCUBE Corporation. *nCUBE 2 Programmers Guide*, *r2.0*, December 1990. Citation on page 2.
- [54] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. Citation on page 643.
- [55] William J. Nitzberg. *Collective Parallel I/O*. PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. Citation on page 643.
- [56] 4.4BSD Programmer's Supplementary Documents (PSD). O'Reilly and Associates, 1994. Citation on page 548.
- [57] Paul Pierce. The NX/2 operating system. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384–390. ACM Press, 1988. Citation on page 2.
- [58] Martin Schulz and Bronis R. de Supinski. P<sup>N</sup>MPI tools: A whole lot greater than the sum of their parts. In ACM/IEEE Supercomputing Conference (SC), pages 1–10. ACM, 2007. Citation on page 730.
- [59] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. Citation on page 643.
- [60] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. Citations on pages 2 and 312.
- [61] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. Citation on page 2.
- [62] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries
   <sup>47</sup> in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable* <sup>48</sup>

3

4

 $\mathbf{5}$ 

6

 $\overline{7}$ 

8

9

10 11

12

13

14

15

16 17

18

19

20

21 22

23

 $^{24}$ 

25 26

27

28

29

30 31

32

33

34

35

36

37 38

39

40

41

42 43

44

45

 $\mathbf{2}$ 

- *Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993. Citation on page 311.
- [63] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University Dept. of Computer Science, August 1994. Archived at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6283. Citation on page 191.
  - [64] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994. Citations on pages 312 and 354.
  - [65] The Internet Society. XDR: External Data Representation Standard, May 2006. http: //www.rfc-editor.org/pdfrfc/rfc4506.txt.pdf. Citation on page 701.
- [66] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing
   sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.
   Citation on page 643.
  - [67] Jesper Larsson Träff. SMP-aware message passing programming. In Eighth International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS), 17th International Parallel and Distributed Processing Symposium (IPDPS), pages 56–65, 2003. Citation on page 343.
- <sup>24</sup> [68] The Unicode Standard, Version 2.0. Addison-Wesley, 1996. ISBN 0-201-48345-9. Ci <sup>25</sup> tation on page 701.
  - [69] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. Citation on page 2.

## **General Index**

!(\_c), <u>839</u>

This index lists mainly terms of the MPI specification. The underlined page numbers refer to the definitions or parts of the definition of the terms. Bold face numbers mark section titles.

\_c, 22, 23, 234, <u>839</u> absolute addresses, 22, <u>141</u>, 827 access epoch, 593action in function names, 12 active, <u>71</u>, 72, 73, 93, 100, 101, 387 active target communication, 593 addresses, 156absolute, **22**, 23, <u>141</u>, 827 correct use, 156 relative displacement, 22, 141 alignment, 456, 557, 560, 1047 all-reduce, 238 nonblocking, 270 persistent, 295 all-to-all, 217 nonblocking, 263 persistent, 289 array arguments, 20 assertions, 607 ASYNCHRONOUS Fortran attribute, 829 attribute, <u>313</u>, <u>365</u>, **849** caching, 312barrier synchronization, 194 nonblocking, 252 persistent, 277 basic datatypes,  $\underline{33}$ additional host language, 34 byte, **33** C, <u>**34</u></u></u>** C and Fortran,  $\underline{35}$ C++, <u>35</u> Fortran, 33 packed, 33blocking, 16, 49, 53, 89, 659 I/O, **660** point-to-point, 32 blocking operation, 14

	11
blocking procedure, <u>16</u>	12
bounds of datatypes, 147	13
broadcast, <b>194</b>	14
nonblocking, <b>253</b>	15
persistent, <b>278</b>	16
buffered mode send, $50$ , 53, 93, 96	17
buffer allocation, 57	18
nonblocking, $\underline{61}$ , $62$ , $64$	
С	19
language binding, 25	20
caching, <b>311</b> , <u>312</u> , <b>365</b>	21
callback functions	22
language interoperability, 848	23
prototype definitions, 872	24
deprecated, 878	25
cancel, 25, 72, <u>84</u> , 84, 85, <b>92</b> , 93, 94, 496, 784,	26
1046	27
cancelled, $\underline{84}$ , $85$ , $92-94$	28
canonical pack and unpack, 182	29
Cartesian	30
topology, <u>391</u> , <b>392</b>	31
Chameleon, $2$	
change-log, <b>1045</b>	32
Chimp, 2	33
choice, 21	34
class	35
in function names, $\underline{12}$	36
clock synchronization, <b>454</b>	37
collective, $\underline{16}$ , $659$	38
split, 709	39
collective communication, 187	40
correctness, <b>301</b> file data access operations, <b>684</b>	41
neighborhood, 416	
nonblocking, 250	42
progress, 251, 306, 308	43
collective operation, <u>15</u>	44
collective procedure, 16	45
commit, $150$	46
COMMON blocks, 832	47
communication, 551	48

1 collective, 187  $\mathbf{2}$ modes. 49 one-sided, 551 3 overlap with communication, 614 overlap with computation, 615partitioned point-to-point, 103 6 point-to-point, 31 7 RMA, <u>551</u> 8 typed, untyped, and packed, 45 9 communication modes, 49 10 communicator, 36, 311, 312 hidden, 190, 304, 601 11 complete operation, 13 12completing,  $\underline{16}$ 13completing procedure, 1614completion, 49, 61, 70, 71, 72, 75, 94 15multiple, **75**, 76–80 16completion stage, 1317connected, 546 constants, 20, 853, 857 18 context, **311**, 312, 314 19 control variables 20tools interface, 738 21conversion, 48 22representation,  $\underline{48}$ 23type, 48 $^{24}$ counts, 22 25create in function names, 122627data, 33 28data conversion, 48 29datatype, 17 30 derived,  $\underline{17}$  $^{31}$ equivalent, 17 named, 1732 portable, 17 33 predefined, 1734 unnamed, 17 35 datatypes, 119, 846 36 deadlock avoidance 37 cyclic shift, 4238 nonblocking communication, 60, 75 send modes, 57 39 default file error, 720 40 delete 41 in function names, 12 42deprecated interfaces, 23, 781 43 derived datatype, <u>17</u>, **119**, **823** 44disconnected, 54645displacement, 643, 657 distributed graph 46topology, 391, 396 47Dynamic Process Model, 487 48

dynamically attached memory, 562 elementary datatype, 643, 658 empty, <u>71</u>, 72, 76 end of file, 645 envelope, 32, 35, 35, 36, 38, 40, 48, 49, 60 data representation conversion. 48 environmental inquiries, 453 equivalent datatypes, 17 erroneous program, 38 freeing active request, 73 invalid matched receive, 91, 92 lack of buffer space, 55 ready mode before receive, 50 type matching, 45 error handling, 26, 458 default file error handler, 646, 649, 720 error codes and classes, 469, 473 error handlers, 461, 473, 848 fatal after request free, 74 finalize, 27, 496, 497, 547 I/O, 719, 720 initial error handler, 27, 459, 489, 497, 531 multiple completions, 78, 81 one-sided communication, 609 process failure, 27, 547 program error,  $\underline{27}$ resource error, 27 startup, 27, 489, 531 transmission failure, 27 establishing communication, 533 etype, 643, 658 events tools interface, 757 examples, 29 exclusive scan nonblocking, 275 persistent, **300** explicit offsets, 660, 662 exposure epoch, 593extent of datatypes, <u>120</u>, <u>145</u>, **147** true extent, 149 external32 file data representation, 699 extra-state, 853 fairness not guaranteed, 55requirement, 81 file, 643

data access, collective operations, explicit offsets, individual file pointers,

seek, 687 shared file pointers, 680 split collective, 688 end of file,  $\underline{645}$ filetype, 644 handle,  $\underline{645}$ interoperability, 697 manipulation, 645nonblocking collective progress, 713 offset, 22, <u>644</u> pointer,  $\underline{645}$ progress, 713 size,  $\underline{645}$ view, 643, 644, 656 file size, 715 finalize, 494 finished, 497 Fortran language binding, 24, 791 Fortran support, 791 freeing procedure, 16freeing stage, <u>14</u>, 73, 83 gather, 196 nonblocking, 254 persistent, 279 gather-to-all, 213 nonblocking, 260 persistent, 286 general datatype, 119generalized requests, 633, 633 progress, 633get in function names,  $\underline{12}$ graph topology, <u>391</u>, **394** group, 36, **311**, <u>312</u>, <u>314</u>, 356 group objects, 314handles, 18, 841 hardware resource type, 339, 340 host rank, 453 immediate, <u>16</u>, 61, 62, 77, 80, 81, 85, 92 inactive, <u>71</u>, 72, 73, 76, 77, 93, 99-101 inclusive scan, 246 nonblocking, 274 persistent, 299 incomplete, <u>16</u>, 62, 89, 90 incomplete procedure,  $\underline{16}$ independent,  $\underline{546}$ individual file pointers, 660, 669 info object, 479

file info, 653

keys, <b>879</b>	1
values, <b>880</b>	2
initial error handler, <u>27</u>	3
initialization, 88, 94, 100	4
initialization procedure, $\underline{15}$	5
initialization stage, $\underline{13}$	6
initiation, <u>13</u> , 61, <b>62</b> , 70, 74, 85	
initiation procedure, $\underline{16}$	7
inter-communication, $\underline{313}$ , $\underline{355}$	8
inter-communicator, $\underline{313}$ , $\underline{355}$	9
collective operations, <b>191</b> , <b>192</b>	10
point-to-point, $36$ , $38$	11
interlanguage communication, $854$	12
internal	13
file data representation, $\underline{698}$	14
interoperability, 697	
intra-communication, $\underline{313}$ , $\underline{355}$	15
intra-communicator, $\underline{312}$ , $\underline{355}$	16
collective operations, <b>190</b>	17
intra-communicator objects, <u>315</u>	18
I/O, <b>643</b>	19
IO rank, <b>454</b>	20
IS	21
in function names, $\underline{12}$	22
language binding, 23, 791	23
interoperability, 840	24
summary, 857	25
large	26
$\begin{array}{c} \text{count, } \underline{23}, 839 \\ \text{dimber compart}, 820 \end{array}$	27
displacement, 839	28
large count, $\frac{23}{125}$ 120, 144, 145, 150	29
lb_marker, 135, 139, <u>144</u> , 145, 150	30
erased, $\underline{148}$	31
local, $\underline{15}$ , 50, 61, 72, 73, 79, 81, 85, 88, 92, 94,	
100, 503 local group, 328	32
local procedure, 15	33
logically concurrent, $54$	34
loosely synchronous model, <b>386</b>	35
lower bound, $\underline{145}$	36
lower-bound markers, 144	37
	38
macros, 26	39
main thread, 518	40
matched probe	
progress, 89	41
matched receive, 88, <b>90</b> , <u>90</u>	42
matching	43
type, <u>153</u> , <b>714</b>	44
matching probe, 86, 87, <u>87</u> , 88–90	45
matching receive, 89	46
matching rules	47
blocking with nonblocking, $61$	48
= '	40

General Index

1	cancel, 93
2	envelope, <u>38</u>
3	null process, <b>101</b>
4	ordering, 54
5	persistent with nonpersistent, $101$
6	probe, $85, 87$
	send modes, $\underline{53}$
7	type, $45, 45$
8	wildcard, $\underline{38}$
9	memory
10	alignment, 456, 557, 560, 1047
11	allocation, <b>455</b> , 556, 558
12	system, $\underline{18}$
13	memory model, 552, <u>592</u>
14	separate, 552, 560
15	unified, 552, 560
16	message, 31, 87
	buffer, 51
17	cancel, $72, \underline{84}, 85, 92-94$
18	data, 31, 32, <b>33</b> , 49, 51
19	envelope, 32, <b>35</b> , <u>35</u> , 36, 38, 40, 48, 49, 60
20	handle, 87, <u>88</u> , 91, 92 intermediate buffering, 45, 49
21	invalid handle, $\underline{89}$
22	predefined handle, $\underline{89}$
23	wildcard, <u>38</u>
24	message handle, 87, <u>88</u> , 91, 92
25	invalid, <u>89</u>
26	predefined, $\underline{89}$
	modes, $49$
27	buffered, $\underline{50}$ , $\underline{53}$
28	ready, $\underline{50}, \underline{53}$
29	standard, $\underline{49}$ , 53
30	synchronous, $50, 53$
31	module variables, 832
32	MPI datatype, <b>17</b>
33	mpi module
34	Fortran support, <b>795</b>
35	MPI operation, 13
36	MPI procedure, <b>15</b>
37	MPI process initialization, $\underline{487}$
38	Dynamic Process Model, 487
	Sessions Model, 315, 318, 453, 487, 519,
39	646
40	World Model, 315, 318, 453, 487, 518, 646,
41	720
42	mpi_f08 module
43	Fortran support, <b>792</b>
44	MPI_SIZEOF and storage_size(), 25, 786, 815–817
45	mpiexec, 490, <u>515</u>
46	mplexec, 490, <u>515</u> mplf.h include file
47	Fortran support, <b>797</b>
48	mpirun, 515
	······································

multiple completions, 75, 76-80 error handling, 78, 81 named datatype, 17names, **534** name publishing, 538 naming objects, 381 native file data representation,  $\underline{698}$ neighborhood collective communication, 416 nonblocking, 429 periodic and dims==1 or 2, 1045non-local, <u>15</u>, 49, 50, 61, 71, 85, 89, 90 non-local procedure, 15 non-overtaking, <u>54</u>, <u>74</u> nonblocking, 16, 60, 73, 88, 92, 569, 659 communication,  $\underline{60}$ completion, **70**, **72** Fortran problems, 826 I/O, 660 initiation, 62 persistent partitioned completion, 110 request objects, 62 nonblocking operation, 14nonblocking procedure,  $\underline{16}$ noncollective operation,  $\underline{15}$ null handle, <u>71</u>, 76–78 null handles, 79, 83

offset, 22, <u>644</u> one-sided communication, 551 Fortran problems, 827 progress, 619, 621 opaque objects, 18, 846 operation, 13 blocking, 14 collective,  $\underline{15}$ complete, 13nonblocking,  $\underline{14}$ noncollective, 15operation-related,  $\underline{15}$ partitioned receive, 14partitioned send, 14persistent, 14 semantics, 13 stage, 13 completion,  $\underline{13}$ freeing, 14 initialization,  $\underline{13}$ starting,  $\underline{13}$ ordered, 54, 74origin, 552

null processes, 101

pack, 174 canonical, 182packing unit, 177 parallel procedure, 386partitioned completion, 110 partitioned point-to-point communication, 103 passive target communication, 593 performance variables tools interface, 744 persistent communication request, 71–73, 76-80, 93, **94**, <u>94</u>, 95-99, 101 active,  $\underline{71}$ completion, 72, 94 inactive, 71starting, 94, 99 persistent communication requests collective persistent, 276, 437 Fortran problems, 827 persistent operation, 14PICL, 2 PMPI\_, 725 point-to-point communication, 31 blocking, **32** buffer allocation, 57 cancel, **92** matched receive, 90 matching probe, 87 nonblocking, 60 persistent, 94 probe, 84 receive operation, 36 send modes, 49, 53 send operation, 32 send-receive operation, 42 status, 39portable data type,  $\underline{17}$ ports, **534** POSIX environment, 477 FORTRAN, 19 I/O, 646, 647, 660, 698, 710 model, 643predefined datatype,  $\underline{17}$ predefined reduction operations, 226 private window copy, 591 probe, 84, 84, 85, 88 matching, 87 progress, 85 procedure, 15 blocking, 16collective, 16completing, 16freeing,  $\underline{16}$ immediate,  $\underline{16}$ 

incomplete, $\underline{16}$	1
initialization, $\underline{15}$	2
initiation, $\underline{16}$	3
local, $\underline{15}$	4
non-local, $\underline{15}$	5
nonblocking, <u>16</u>	
operation-related, $\underline{15}$	6
semantics, 15	7
specification, 12	8
starting, $15$	9
synchronizing, $17$	10
procedure specification, 12	11
process creation, 487	12
process failures, $27$	
process set names, 879	13
processes, 26	14
processor name, 454	15
profiling interface, <b>725</b>	16
program error, $\underline{27}$	17
progress	18
file, 713	19
nonblocking collective, 713	
finalize	20
buffered data, 496	21
generalized requests, 633	22
nonblocking collective communication,	23
251, 306, 308	24
one-sided communication, <b>619</b> , 621	25
partitioned point-to-point communication,	
111	26
point-to-point communication, $54$ , $75$ , $85$ ,	27
89	28
threads, 518	29
prototype definitions, 872	30
deprecated, 878	31
	32
public window copy, 591	
PVM, 2	33
rank, <u>314</u>	34
ready mode send, $50$ , $53$ , $98$ , $100$	35
as standard mode send, $53$	36
nonblocking, $\underline{61}$ , $\underline{62}$ , $\underline{66}$	37
receive, $31$ , $32$	38
blocking, <b>36</b> , <u>36</u>	39
buffer, 32, 90, 99	
complete, $\underline{61}$	40
context, 356	41
	42
matched, $88$ , $90$ , $90$	43
nonblocking, $61, 67$	44
start, $\underline{61}$ , $67$	45
reduce, <b>224</b> nonblocking, <b>269</b>	46
persistent, <b>294</b> reduce-scatter, <b>242</b>	47
icult-statici, 444	48

1 nonblocking, 271, 273  $\mathbf{2}$ persistent, 296, 298 reduction operations, 223, 848 3 predefined. 226 4 process-local, 240 5scan, **245** 6 user-defined, 233 7 related, 1778 relative displacement, 22, 141 9 remote group, 328 10 Remote Memory Access see RMA, <u>551</u> 11 removed interfaces, 23, 787 12representation 13 conversion, 4814request complete 15I/O, 660 16request objects, 62 17completion, 75 freeing, 73, 101 18 initiation, 74 19 multiple completions, 75 20null handle, 71, 76–78 21null handles, 79, 83 22 started, 49, 75, 89, 94, 99, 101 23resource error, 27  $^{24}$ RMA, 551 communication calls, 569 25request-based, 584 26memory model, 591 27synchronization calls, 593 2829scan, **245** 30 inclusive, 246  $^{31}$ scatter, 206 nonblocking, 257 32 persistent, 283 33 seek, 687 34 semantic changes, 789 35 semantics, 13, 1046 36 collective communications, 188 37 nonblocking progress, 251, 306, 308 38 data conversion. 48 deadlock 39 buffer space, 5640 cyclic shift, 4241 send to self, 3842erroneous program 43 freeing active request, 73 44 invalid matched receive, 91, 92 45lack of buffer space, 55 ready mode before receive, 50 46 type matching, 4547exceptions 48

completing and local, 51incomplete and non-local, 90 file collective, 713 collective access, 684 conflicting access, 710 consistency, 709 explicit offsets, 662 nonblocking collective, 713 progress, 713 shared file pointer, 680split collective, 709 finalize buffered data, 496 generalized requests progress, 633 inter-communicator. 328 MPI\_COMM\_IDUP, 330 nonblocking communications, 74 nonblocking completion, 70 nonblocking partitioned communications, 111 operation, 13 overlap, 61 partitioned point-to-point communication, 104 progress, 111 point-to-point communication, 54 cancel, 72, <u>84</u>, 84, 85, **92**, 93, 94 deterministic, 54 fairness not guaranteed, 55fairness requirement, 81logically concurrent, 54 matched receive, 90 matching probe, 87 non-overtaking, <u>54</u>, <u>74</u> ordered, <u>54</u>, <u>74</u> overflow error on trunction, 37persistent, 94 probe, 84 progress, 54, 75, 85, 89 resource limitations, 55 send modes, **49**, 53 send-receive concurrency, 43 wildcard receive, 38 procedure, 15 process failure, 547 terms, 13threads progress, 518 semantics and correctness one-sided communication, 610 progress, 619, 621 send, 31

blocking, **32**, <u>32</u> buffer, 31 complete,  $\underline{61}$ context, 356 nonblocking, 61 start, <u>61</u>, 63–66 send-receive blocking, **42**, <u>42</u> nonblocking, 69, 70 start, 69, 70 separate memory model, 552, 560, 592 sequential storage,  $\underline{156}$ serialization, 61 Sessions Model, 315, 318, 453, 487, 519, 646  $\operatorname{set}$ in function names,  $\underline{12}$ shared file pointers, <u>660</u>, **680** shared memory allocation, 558 signals, 29 singleton init, 544 size changing I/O, 715 source, 356 split collective, 659, 688 stage,  $\underline{13}$ completion,  $\underline{13}$ freeing,  $\underline{14}$ initialization, 13starting,  $\underline{13}$ standard mode send,  $\underline{49}$ , 53, 95 as synchronous mode send, 53nonblocking, 61, 63 started, 497 request objects, 49, 75, 89, 94, 99, 101 starting procedure, 15, 94 starting processes, 521, 522 starting stage,  $\underline{13}$ startup, 488 portable, 515 state, **20** status, 38, 843 array in Fortran, 39 associating information, 640 derived type in Fortran 2008, 39 empty, <u>71</u>, 72, 76–79, 83 error in status, 39 for send operation, 71 ignore, **41** message length, 39structure in C, 39test, 83 strong synchronization, 594synchronization, 551, 569 synchronization calls

RMA, <b>593</b>	1
synchronizing procedure, <u>17</u>	2
synchronous mode send, <u>50</u> , 53, 70, 89, 97	3
nonblocking, <u>61</u> , 62, 65	4
system memory, $\underline{18}$	5
	6
tag values, <b>453</b>	
target, $\underline{552}$	7
thread compliant, $\underline{492}$ , $\underline{517}$	8
threads, <b>517</b>	9
progress, 518	10
thread-safe, 2, 452, 469, 470, 761	11
timers and synchronization, 477	12
tool information interface, 731	13
tool support, <b>725</b>	14
topologies, 389	
topology	15
Cartesian, <u>391</u> , <b>392</b>	16
distributed graph, $\underline{391}$ , $396$	17
$\operatorname{graph}, \frac{391}{222}, 394$	18
virtual, <b>390</b>	19
transmission failures, $\underline{27}$	20
true extent of datatypes, <b>149</b>	21
TS 29113, 21, 24, 791–796, 798–801, 804–808,	22
817-819, 824, 829, 832, 834, 1055	
type	23
conversion, $\frac{48}{10}$	24
matching rules, <b>45</b>	25
type map, $\underline{120}$ type matching, $\underline{152}$	26
type matching, $\underline{153}$	27
type signature, $\underline{120}$	28
types, <b>871</b>	29
ub_marker, 135, 139, 140, <u>144</u> , 145, 150	30
erased, $\underline{148}$	31
unified memory model, $552$ , $560$ , $592$	32
universe size, 544	
unnamed datatype, <u>17</u>	33
unpack, <b>174</b>	34
canonical, 182	35
upper bound, $\underline{145}$	36
upper-bound markers, 144	37
user functions at process termination, 498	38
user-defined data representations, <b>704</b>	39
user-defined reduction operations, 233	40
	41
verbosity levels	
tools interface, 732	42
version inquiries, <b>451</b>	43
view, 643, <u>644</u> , <b>656</b>	44
virtual topology, <u>312, 313</u> , <b>390</b>	45
weak synchronization 505	46
weak synchronization, <u>595</u>	47
wildcard, <u>38</u> window	48
willdow	

1 2	allocation, <b>556</b> creation, <b>553</b>
3	dynamically attached memory, <b>562</b> shared memory allocation, <b>558</b>
4	World Model, 315, 318, 453, 487, 518, 646, 720
5 6	XDR, 49
7	Zipcode, 2
8 9	Zipcoue, Z
10	
11	
12	
13	
14	
15	
16 17	
18	
19	
20	
21	
22	
23	
24	
25 26	
20	
28	
29	
30	
31	
32	
33	
34 35	
36	
37	
38	
39	
40	
41	
42	
43 44	
44	
46	
47	
48	

## **Examples Index**

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C examples.

ASYNCHRONOUS, 626, 835, 837 Attributes between languages, 849	
Blocking/Nonblocking collectives do not match, 306	
C/Fortran handle conversion, 843 Cartesian virtual topologies, 445 Client-server code, 82 with probe, 86 with probe (wrong), 86	
Datatype 3D array, 166 absolute addresses, 171 array of structures, 168 elaborate example, 179, 180 matching type, 153 matrix transpose, 167 union, 172 Datatypes matching, 46 not matching, 46 untyped, 46	
Deadlock if not buffered, 56 with MPI_Bcast, 301, 302 wrong message exchange, 56	
False matching of collective operations, 305 Fortran 90 copying and sequence problem, 819, 821 822 derived types, 823 heterogeneous communication (unsafe), 816, 817 invalid KIND, 812 MPI_TYPE_MATCH_SIZE implementation, 815 overlapping communication and computation, 836, 838	Ļ,

register optimization, 826, 828	13
Independence of penklesking energy 200	14
Independence of nonblocking operations, 308 Inter-communicator, 332, 338	15
Interlanguage communication, 854	16
Intertwined matching pairs, 55	17
meet whice matching pairs, 55	18
Message exchange (ping-pong), 56	19
Mixing blocking and nonblocking collective	20
operations, $304$	21
Mixing collective and point-to-point requests,	21
307	
MPI_ACCUMULATE, 578	23
MPI_Accumulate, 624, 625, 628	24
MPI_Aint, 168	25
MPI_Aint_add, 628	26
MPI_Allgather, 216	27
MPI_ALLOC_MEM, 457, 458	28
MPI_Alloc_mem, 458, 628	29
MPI_ALLREDUCE, 240	30
MPI_Alltoall, 306 MPI_ASYNC_PROTECTS_NONBLOCKING,	31
626	32
MPI_BARRIER, 511	33
MPI_Barrier, 496, 614–618, 623–626	
MPI_Bcast, 195, 301–305	34
MPI_BSEND, 54, 55	35
MPI_Buffer_attach, 59, 496	36
MPI_Buffer_detach, 59	37
MPI_BYTE, 46	38
MPI_Cancel, 496	39
MPI_CART_COORDS, 413	40
MPI_CART_GET, 445	41
MPI_CART_RANK, 413	42
MPI_CART_SHIFT, 413, 424, 445	43
MPI_CART_SUB, 414	40
$MPI\_CARTDIM\_GET, 424$	
MPI_CHARACTER, 47	45
MPI_Comm_accept, 541, 542	46
MPI_Comm_connect, 541, 542	47
MPI_Comm_create, 332, 349, 350, 353	48

1	MPI_COMM_CREATE_FROM_GROUP, 507,
2	511
3	MPI_Comm_create_keyval, 379
4	MPI_Comm_dup, 352
5	MPI_Comm_get_attr, 379
	MPI_Comm_get_parent, 531
6	MPI_Comm_group, 332, 353, 379
7	MPI_Comm_rank, 31
8	MPI_Comm_remote_size, 338
9	MPI_Comm_set_attr, 379
10	MPI_COMM_SPAWN, 525
11	MPI_Comm_spawn, 525, 531
12	MPI_COMM_SPAWN_MULTIPLE, 530
	MPI_Comm_spawn_multiple, 530
13	MPI_Comm_split, 338, 362, 363
14	MPI_Comm_split_type, 341, 343
15	MPI_Compare_and_swap, 626, 628
16	MPI_DIMS_CREATE, 394, 445
17	MPI_DIST_GRAPH_CREATE, 401
18	MPI_Dist_graph_create, 402
19	MPI_DIST_GRAPH_CREATE_ADJACENT,
20	401
	MPI_F_sync_reg, 626
21	MPI_FILE_CLOSE, 671, 675
22	MPI_FILE_GET_AMODE, 652
23	MPI_FILE_IREAD, 675
24	MPI_FILE_OPEN, 671, 675
25	MPI_FILE_READ, 671
26	MPI_FILE_SET_ATOMICITY, 715
27	MPI_FILE_SET_VIEW, 671, 675
28	MPI_FILE_SYNC, 716
29	MPI_Finalize, 495–497
	MPI_FREE_MEM, 457, 458
30	MPI_Free_mem, 628
31	MPI_Gather, 180, 200, 201, 205
32	MPI_Gatherv, 180, 201–205
33	MPI_GET, 574, 576
34	MPI_Get, 614–617, 622, 623
35	MPI_Get_accumulate, 624, 625, 628
36	MPI_GET_ADDRESS, 142, 823, 824, 846
37	MPI_Get_address, 168, 171, 172, 179
38	MPI_GET_COUNT, 155
	MPI_GET_ELEMENTS, 155
39	MPI_GRAPH_CREATE, 395, 408, 409
40	MPI_GRAPH_NEIGHBORS, 408
41	MPI_GRAPH_NEIGHBORS_COUNT, 408
42	MPI_Grequest_complete, 638
43	MPI_Grequest_start, 638
44	MPI_Group_excl, 349 MPI_Group_free_222_240_250
45	MPI_Group_free, 332, 349, 350
46	MPI_GROUP_FROM_SESSION_PSET, 507,
47	510, 511 MPL Croup ind 232, 250, 252
	MPI_Group_incl, 332, 350, 353 MPI_Iallreduce, 307
48	$1111_1ameduce, 307$

MPI\_Ialltoall, 306 MPI\_Ibarrier, 304–307 MPI\_Ibcast, 254, 307, 308 MPI\_INFO\_CREATE, 507 MPI\_INFO\_ENV, 490 MPI\_INFO\_SET, 507 MPI\_Init, 31 MPI\_Intercomm\_create, 362, 363 MPI\_Iprobe, 496 MPI\_IRECV, 73-75, 82 MPI\_Irecv, 307 MPI\_ISEND, 73, 74, 82 MPI\_NEIGHBOR\_ALLGATHER, 420 MPI\_NEIGHBOR\_ALLTOALL, 424 MPI\_Op\_create, 237, 238, 248 MPI\_Open\_port, 541, 542 MPI\_Pack, 179, 180 MPI\_Pack\_size, 180 MPI\_Parrived, 115 MPI\_Pready, 104, 112, 113, 115 MPI\_Precv\_init, 104, 112, 113, 115 MPI\_PROBE, 86 MPI\_Psend\_init, 104, 112, 113, 115 MPI\_Publish\_name, 542 MPI\_Put, 599, 605, 615, 616, 618, 622, 623 MPI\_RECV, 46, 47, 54-56, 75, 86, 153 MPI\_Recv, 31, 306 MPI\_REDUCE, 228, 231 MPI\_Reduce, 231, 232, 237, 238 MPI\_REQUEST\_FREE, 74 MPI\_Request\_free, 496 MPI\_Rget, 627 MPI\_Rput, 627 MPI\_Scan, 248  $MPI\_Scatter, 210$ MPI\_Scatterv, 210, 211 MPI\_SEND, 46, 47, 56, 75, 86, 153 MPI\_Send, 31, 168, 171, 172, 179, 306, 307 MPI\_SENDRECV, 166, 167, 424 MPI\_SENDRECV\_REPLACE, 413 MPI\_SESSION\_FINALIZE, 507, 510, 511 MPI\_SESSION\_GET\_NTH\_PSET, 510, 511 MPI\_SESSION\_GET\_NUM\_PSETS, 510, 511 MPI\_SESSION\_INIT, 507, 510, 511 MPI\_SSEND, 55, 75 MPI\_Test\_cancelled, 496 MPI\_TYPE\_COMMIT, 151, 166, 167, 574, 823, 824 MPI\_Type\_commit, 168, 171, 172, 179, 201-205, 211, 248 MPI\_TYPE\_CONTIGUOUS, 122, 145, 153, 155MPI\_Type\_contiguous, 201 MPI\_TYPE\_CREATE\_DARRAY, 140

MPI\_TYPE\_CREATE\_HVECTOR, 166, 167 MPI\_Type\_create\_hvector, 168, 171 MPI\_TYPE\_CREATE\_INDEXED\_BLOCK, 574MPI\_TYPE\_CREATE\_RESIZED, 823, 824 MPI\_TYPE\_CREATE\_STRUCT, 132, 145, 167, 823, 824 MPI\_Type\_create\_struct, 168, 171, 172, 179, 203, 205, 248 MPI\_TYPE\_CREATE\_SUBARRAY, 723 MPI\_TYPE\_EXTENT, 574 MPI\_TYPE\_FREE, 574 MPI\_Type\_get\_contents, 173 MPI\_Type\_get\_envelope, 173 MPI\_TYPE\_GET\_EXTENT, 166, 167, 576, 578 $MPI_Type_get_extent, 168$ MPI\_TYPE\_INDEXED, 127, 166 MPI\_Type\_indexed, 168, 171 MPI\_TYPE\_VECTOR, 123, 166, 167 MPI\_Type\_vector, 201, 202, 204, 211 MPI\_Unpack, 179, 180 MPI\_Unpublish\_name, 542 MPI\_User\_function, 238 MPI\_WAIT, 73-75, 82, 675 MPI\_Wait, 304–307 MPI\_Waitall, 307, 627 MPI\_WAITANY, 82 MPI\_Waitany, 627 MPI\_WAITSOME, 82 MPI\_Win\_attach, 628 MPI\_Win\_complete, 599, 617, 618, 623 MPI\_WIN\_CREATE, 574, 576, 578 MPI\_Win\_create\_dynamic, 628 MPI\_Win\_detach, 628 MPI\_WIN\_FENCE, 574, 576, 578 MPI\_Win\_fence, 622 MPI\_Win\_flush, 616, 624-626, 628 MPI\_Win\_flush\_all, 625 MPI\_Win\_flush\_local, 615 MPI\_WIN\_FREE, 576, 578 MPI\_Win\_lock, 605, 614-618 MPI\_Win\_lock\_all, 626-628 MPI\_Win\_post, 617, 618, 623 MPI\_Win\_start, 599, 617, 618, 623 MPI\_Win\_sync, 615, 616, 624-626 shared memory windows, 626 MPI\_Win\_unlock, 605, 614-618 MPI\_Win\_unlock\_all, 627, 628 MPI\_Win\_wait, 617, 618, 623 mpiexec, 490, 516, 517

Neighborhood collective communication, 445 Non-overtaking messages, 54

Nonblocking operations, 73, 74	1
message ordering, 74	2
progress, 75	3
Nondeterministic program with MPI_Bcast,	4
303	5
Overlanning Communicators 207	6
Overlapping Communicators, 307	7
Partitioned Communication	8
Equal send/recv partitioning, $112$	9
Partial completion notification, $115$	10
Send with tasks, $113$	11
Simple example, 104	12
Pipelining nonblocking collective operations,	13
307 Beint to peint	14
Point-to-point Hello world, 31	15
Profiling interface	16
implementation using the C macro	10
preprocessor, 729	18
implementation using weak symbols, 728	
measurement wrapper, 728	19
Progress of matching pairs, 55	20
Progression of nonblocking collective	21
operations, $306$	22
Shared memory windows	23
MPI_Win_sync, 626	24
	25
Threads and MPI, 518	26
Tool information interface	27
basic usage of performance variables, 755	28
listing names of all control variables, 741	29
reading the value of a control variable, 744 Topologies, 445	30
Typemap, 121–123, 127, 132, 140	31
1) pointap, 121 120, 121, 102, 110	32
Virtual topologies, 445	33
	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	45
	46
	47
	48

# MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles, including info keys and values. Underlined page numbers give the location of the primary definition or use of the indexed term.

14"access\_style", 655, 879 15"accumulate\_ops", 554, 879 16"accumulate\_ordering", 554, 619, 879 "alloc\_shared\_noncontig", 559, 560, 879 17"appnum", 545, 879 18 "arch", 490, 530, 879 19 "argv", 490, 879 2021"cb\_block\_size", 655, 879 22 "cb\_buffer\_size", 655, 879 "cb\_nodes", 655, 879 23"chunked", 655, 880 24"chunked\_item", 655, 879 25"chunked\_size", 655, 880 26"collective\_buffering", 655, 880 27"command", 490, 880 28"external32", 182, 698, 699, 701-704, 709, 29811-814, 817, 879, 1052, 1059 30 31"false", 346, 655, 880 32 "file", 490, 531, 880 33 "file\_perm", 655, 880 34"filename", 655, 880 35 "host", 490, 530, 880 36 "hwloc://L3Cache", 503 37 38 "internal", 698, 699, 709, 879 39 "io\_node\_list", 656, 880 "ip\_address", 541, 880 40"ip\_port", 541, 880 41 42"maxprocs", 490, 491, 880 43"mpi://", 503, 879 44"mpi://SELF", 503, 504, 646, 879 45"mpi://WORLD", 503, 504, 509, 515, 545, 879 MPI::\_LONG\_LONG, 1052 46 MPI::BOOL, 1052 47MPI::COMPLEX, 1052 48

5 6

12

13

MPI::DOUBLE\_COMPLEX, 1052 MPI::F\_COMPLEX16, 1052 MPI::F\_COMPLEX32, 1052 MPI::F\_COMPLEX4, 1052 MPI::F\_COMPLEX8, 1052 MPI::INTEGER16, 1052 MPI::LONG\_DOUBLE\_COMPLEX, 1052 MPI::LONG\_LONG, 1052 MPI::REAL16, 1052 MPI\_2DOUBLE\_PRECISION, 230, 863 MPI\_2INT, 230, 863 MPI\_2INTEGER, 230, 863 MPI\_2REAL, 230, 863 MPI\_ADDRESS\_KIND, 21, 22, 35, 365, 818, 849.860 MPI\_AINT, 35, 121, 227, 563, 702, 839, 861, 862, 1057-1060 MPI\_ANY\_SOURCE, 37, 38, 42, 44, 54, 67-69, 71, 84, 85, 87-89, 99, 346, 387, 454, 859 MPI\_ANY\_TAG, 20, 37, <u>38</u>, 40, 42, 44, 67–69, 71, 84, 85, 87–89, 91, 92, 99, 101, 346, 859, 1054 MPI\_APPNUM, 545, 546, 866 MPI\_ARGV\_NULL, 21, 524, 525, 529, 818, 868 MPI\_ARGVS\_NULL, 21, 529, 818, 868 "mpi\_assert\_allow\_overtaking", 346, 880, 1047 "mpi\_assert\_exact\_length", 346, 880, 1047 "mpi\_assert\_no\_any\_source", 346, 880, 1047 "mpi\_assert\_no\_any\_tag", 346, 880, 1047 MPI\_ASYNC\_PROTECTS\_NONBLOCKING. 21, 626, 792, 793, 795, 797, 800, 808, 809, 829, 860, 1056 MPI\_BAND, 226, 228, 864 MPI\_BOR, 226, 228, 864 MPI\_BOTTOM, 12, 21, 22, 41, 112, 141, 156, 157, 191, 398, 400, 527, 563, 567, 794, 796, 804, 818, 823, 825, 827, 828, 830-832, 834, 846-848, 853, 859, 1063

MPI\_BSEND\_OVERHEAD, 60, 859 MPI\_BXOR, 226, 228, 864 MPI\_BYTE, 33, 34, 45-48, 182, 227, 644, 698, 699, 702, 714, 854, 861, 862, 1060 MPI\_C\_BOOL, 34, 227, 702, 861, 1052, 1057 - 1060MPI\_C\_COMPLEX, 34, 227, 702, 861, 1052, 1057 - 1060MPI\_C\_DOUBLE\_COMPLEX, 34, 227, 702, 861, 1057-1060 MPI\_C\_FLOAT\_COMPLEX, <u>34</u>, 227, 702, 861, 1057 - 1060MPI\_C\_LONG\_DOUBLE\_COMPLEX, <u>34</u>, 227, 702, 861, 1057–1060 MPI\_CART, 403, 865 MPI\_CHAR, 34, 48, 132, 229, 702, 736, 861, 1057MPI\_CHARACTER, <u>33</u>, 47, 48, 229, 702, 862 MPI\_COMBINER\_CONTIGUOUS, 159, 163, 867 MPI\_COMBINER\_DARRAY, 159, 165, 867 MPI\_COMBINER\_DUP, 159, 163, 867 MPI\_COMBINER\_F90\_COMPLEX, 159, 165, 867 MPI\_COMBINER\_F90\_INTEGER, 159, 165, 867 MPI\_COMBINER\_F90\_REAL, 159, 165, 867 MPI\_COMBINER\_HINDEXED, 25, 159, 164, 867 MPI\_COMBINER\_HINDEXED\_BLOCK, 159, 164, 867, 1054 MPI\_COMBINER\_HINDEXED\_INTEGER, 25, <u>788</u>, 1052 MPI\_COMBINER\_HVECTOR, 25, 159, 164, 867 MPI\_COMBINER\_HVECTOR\_INTEGER, 25, 788, 1052 MPI\_COMBINER\_INDEXED, 159, 164, 867 MPI\_COMBINER\_INDEXED\_BLOCK, 159, 164, 867 MPI\_COMBINER\_NAMED, 159, 163, 867 MPI\_COMBINER\_RESIZED, 159, 165, 867 MPI\_COMBINER\_STRUCT, 25, 159, 164, 867 MPI\_COMBINER\_STRUCT\_INTEGER, 25, 788, 1052 MPI\_COMBINER\_SUBARRAY, 159, 165, 867 MPI\_COMBINER\_VECTOR, 159, 163, 867 MPI\_COMM\_DUP\_FN, 25, 368, 865, 1056 MPI\_COMM\_NULL, <u>315</u>, 331, 332, 334-336, 339-341, 344, 345, 361, 383, 393, 395, 527, 547-549, 842, 864, 1061 MPI\_COMM\_NULL\_COPY\_FN, 25, 368, 794, 848, 865, 1056 MPI\_COMM\_NULL\_DELETE\_FN, 25, 369, 865 "MPI\_COMM\_PARENT", 383, 879 MPI\_COMM\_SELF, 27, <u>315</u>, 334, 347, 365, 383, 459, 489, 494, 497, 499, 547, 646, 789,

1 863, 1046, 1059 "MPI\_COMM\_SELF", 383, 879 2 MPI\_COMM\_TYPE\_HW\_GUIDED, 339, 340, 3 863, 1047 MPI\_COMM\_TYPE\_HW\_UNGUIDED, <u>340</u>, 5341, 342, 863, 1046 6 MPI\_COMM\_TYPE\_SHARED, 339, 340, 863, 7 1054 8 MPI\_COMM\_WORLD, 20, 28, 36, 313, 315-318, 326, 327, 341-343, 347, 350, 9 359, 383, 393, 453, 454, 462, 473, 10 487-490, 492-495, 497, 499, 514, 515, 11 521-524, 527, 529, 544-547, 697, 719, 12720, 742, 751, 789, 841, 853, 863, 13 1046, 1062 14"MPI\_COMM\_WORLD", 383, 879 15MPI\_COMPLEX, 33, 227, 701, 702, 810, 862 16MPI\_COMPLEX16, 35, 227, 703, 862 MPI\_COMPLEX32, 35, 227, 703, 862 17MPI\_COMPLEX4, <u>35</u>, 227, 703, 862 18 MPI\_COMPLEX8, 35, 227, 703, 862 19MPI\_CONGRUENT, 327, 357, 863 20MPI\_CONVERSION\_FN\_NULL, 708, 865 21MPI\_CONVERSION\_FN\_NULL\_C, 708, 839, 22865, 1046 23MPI\_COUNT, 35, 121, 227, 235, 702, 736, 839, 861, 862, 1053  $^{24}$ MPI\_COUNT\_KIND, 21, 22, 35, 860 25MPI\_CXX\_BOOL, 35, 227, 702, 703, 862, 1051 26MPI\_CXX\_DOUBLE\_COMPLEX, <u>35</u>, 227, 702, 27703, 862, 1051 28 MPI\_CXX\_FLOAT\_COMPLEX, <u>35</u>, 227, 702, 29703, 862, 1051 30 MPI\_CXX\_LONG\_DOUBLE\_COMPLEX, 35,  $^{31}$ 227, 702, 703, 862, 1052 MPI\_DATATYPE\_NULL, 152, 864 32 MPI\_DISPLACEMENT\_CURRENT, 657, 868, 33 1063 34 MPI\_DIST\_GRAPH, 403, 865, 1059 35 MPI\_DISTRIBUTE\_BLOCK, 137, 138, 868 36 MPI\_DISTRIBUTE\_CYCLIC, 137, 138, 868 37 MPI\_DISTRIBUTE\_DFLT\_DARG, 137, 138, 868 38 MPI\_DISTRIBUTE\_NONE, 137, 138, 868 MPI\_DOUBLE, 34, 227, 702, 736, 745-747, 809, 39 861 40 MPI\_DOUBLE\_COMPLEX, <u>34</u>, 227, 701, 702, 41 810, 862 42MPI\_DOUBLE\_INT, 230, 231, 863 43 MPI\_DOUBLE\_PRECISION, <u>33</u>, 227, 702, 810, 44862 45MPI\_DUP\_FN, 25, 368, 782, 866 46MPI\_ERR\_ACCESS, 471, 649, 721, 858 MPI\_ERR\_AMODE, 471, 647, 721, 858 47MPI\_ERR\_ARG, 471, 857 48

1 MPI\_ERR\_ASSERT, 471, 610, 858  $\mathbf{2}$ MPI\_ERR\_BAD\_FILE, 471, 721, 858 MPI\_ERR\_BASE, 457, 471, 610, 858 3 MPI\_ERR\_BUFFER, 471, 857 4 MPI\_ERR\_COMM, 471, 857  $\mathbf{5}$ MPI\_ERR\_CONVERSION, 471, 708, 721, 858 6 MPI\_ERR\_COUNT, 471, 857  $\overline{7}$ MPI\_ERR\_DIMS, 471, 857 8 MPI\_ERR\_DISP, 471, 610, 858 9 MPI\_ERR\_DUP\_DATAREP, 471, 705, 721, 858 MPI\_ERR\_FILE, 471, 721, 858 10 MPI\_ERR\_FILE\_EXISTS, 471, 721, 858 11 MPI\_ERR\_FILE\_IN\_USE, 471, 649, 721, 858 12MPI\_ERR\_GROUP, 471, 857 13MPI\_ERR\_IN\_STATUS, 39, 41, 71, 78, 80, 461, 14470, 471, 637, 662, 858 15MPI\_ERR\_INFO, 471, 858 16MPI\_ERR\_INFO\_KEY, 471, 481, 858 17MPI\_ERR\_INFO\_NOKEY, 471, 481, 858 18MPI\_ERR\_INFO\_VALUE, 471, 481, 858 MPI\_ERR\_INTERN, 460, 471, 857 19MPI\_ERR\_IO, 471, 721, 858 20MPI\_ERR\_KEYVAL, 379, 471, 858 21MPI\_ERR\_LASTCODE, 470, 472, 473, 475, 778, 22859 23MPI\_ERR\_LOCKTYPE, 471, 610, 858  $^{24}$ MPI\_ERR\_NAME, 471, 541, 858 25MPI\_ERR\_NO\_MEM, 456, 471, 858 MPI\_ERR\_NO\_SPACE, 471, 721, 858 26MPI\_ERR\_NO\_SUCH\_FILE, 471, 649, 721, 858 27MPI\_ERR\_NOT\_SAME, 471, 721, 858 28MPI\_ERR\_OP, 472, 609, 857 29MPI\_ERR\_OTHER, 470, 472, 857 30MPI\_ERR\_PENDING, 78, 472, 857  $^{31}$ MPI\_ERR\_PORT, 472, 538, 858 32MPI\_ERR\_PROC\_ABORTED, 472, 514, 858, 1047 33 MPI\_ERR\_QUOTA, 472, 721, 858 34MPI\_ERR\_RANK, 472, 609, 857 35MPI\_ERR\_READ\_ONLY, 472, 721, 858 36 MPI\_ERR\_REQUEST, 472, 857 37MPI\_ERR\_RMA\_ATTACH, 472, 610, 858 38MPI\_ERR\_RMA\_CONFLICT, 472, 610, 858 39MPI\_ERR\_RMA\_FLAVOR, 472, 561, 610, 858 MPI\_ERR\_RMA\_RANGE, 472, 610, 858 40MPI\_ERR\_RMA\_SHARED, 472, 610, 858  $^{41}$ MPI\_ERR\_RMA\_SYNC, 472, 610, 858 42MPI\_ERR\_ROOT, 472, 857 43MPI\_ERR\_SERVICE, 472, 540, 858 44MPI\_ERR\_SESSION, 472, 858, 1048 45MPI\_ERR\_SIZE, 472, 610, 858 46MPI\_ERR\_SPAWN, 472, 525, 526, 858 47MPI\_ERR\_TAG, 472, 857 MPI\_ERR\_TOPOLOGY, 472, 857 48

MPI\_ERR\_TRUNCATE, 472, 857 MPI\_ERR\_TYPE, 159, 161, 472, 720, 857 MPI\_ERR\_UNKNOWN, 470, 472, 857 MPI\_ERR\_UNSUPPORTED\_DATAREP, 472, 721.858 MPI\_ERR\_UNSUPPORTED\_OPERATION, 472, 721,858 MPI\_ERR\_VALUE\_TOO\_LARGE, 472, 706, 858, 1046 MPI\_ERR\_WIN, 472, 610, 858 MPI\_ERRCODES\_IGNORE, 21, 526, 527, 818, 868 MPI\_ERRHANDLER\_NULL, 469, 864 MPI\_ERROR, 39, 71, 250, 584, 844, 860, 1049, 1055 MPI\_ERRORS\_ABORT, 459, 489, 531, 860, 1046 "mpi\_errors\_abort", 531, 880, 1048 MPI\_ERRORS\_ARE\_FATAL, 459, 460, 476, 531, 609, 719, 860, 1046 "mpi\_errors\_are\_fatal", 531, 880, 1048 MPI\_ERRORS\_RETURN, 459, 460, 476, 514, 531, 720, 853, 860 "mpi\_errors\_return", 531, 880, 1048 MPI\_F08\_STATUS\_IGNORE, 845, 869, 1056 MPI\_F08\_STATUSES\_IGNORE, 845, 869, 1056 MPI\_F\_ERROR, 844, 860, 1045 MPI\_F\_SOURCE, 844, 860, 1045 MPI\_F\_STATUS\_IGNORE, 844, 869 MPI\_F\_STATUS\_SIZE, 21, 844, 860, 1045 MPI\_F\_STATUSES\_IGNORE, 844, 869 MPI\_F\_TAG, 844, 860, 1045 MPI\_FILE\_NULL, 648, 720, 864 MPI\_FLOAT, <u>34</u>, 132, 225, 227, 700, 702, 861 MPI\_FLOAT\_INT, 17, 230, 231, 863 MPI\_GRAPH, 403, 865 MPI\_GROUP\_EMPTY, <u>314</u>, 320, 321, 331, 332, 334, 344, 361, 864 MPI\_GROUP\_NULL, 314, 324, 325, 864 MPI\_HOST, 453, 863 "mpi\_hw\_resource\_type", 340, 341, 880, 1047 MPI\_IDENT, <u>318</u>, 327, 863 MPI\_IN\_PLACE, 21, 190, 220, 797, 818, 859 MPI\_INFO\_ENV, 484, 490, 491, 499, 863, 1055 MPI\_INFO\_NULL, 340, 401, 456, 484, 526, 536, 647, 649, 658, 864 "mpi\_initial\_errhandler", 489, 490, 531, 880, 1048 MPI\_INT, 17, <u>34</u>, 120, 226, 700-702, 735, 736, 739, 745, 749, 809, 853, 855, 861 MPI\_INT16\_T, 34, 227, 702, 861, 1057-1060 MPI\_INT32\_T, 34, 227, 702, 736, 861, 1057 - 1060

MPI\_INT64\_T, <u>34</u>, 227, 702, 736, 861, 1057 - 1060MPI\_INT8\_T, 34, 227, 702, 861, 1057-1060 MPI\_INTEGER, 33, 45, 227, 702, 809, 810, 855, 862 MPI\_INTEGER1, 35, 227, 703, 862 MPI\_INTEGER16, 227, 703, 862 MPI\_INTEGER2, 35, 227, 701, 703, 862 MPI\_INTEGER4, 35, 227, 703, 862 MPI\_INTEGER8, 35, 227, 703, 814, 862 MPI\_INTEGER\_KIND, 21, 860 MPI\_IO, <u>453</u>, 454, 863 MPI\_KEYVAL\_INVALID, 369, 370, 371, 859 MPI\_LAND, 226, 227, 864 MPI\_LASTUSEDCODE, 473, 474, 866 MPI\_LB, 25, 788, 1052 MPI\_LOCK\_EXCLUSIVE, 602, 859 MPI\_LOCK\_SHARED, 602, 603, 859 MPI\_LOGICAL, 33, 227, 702, 862 MPI\_LONG, 34, 226, 702, 861 MPI\_LONG\_DOUBLE, 34, 227, 702, 861 MPI\_LONG\_DOUBLE\_INT, 230, 863 MPI\_LONG\_INT, 230, 231, 863 MPI\_LONG\_LONG, 34, 227, 861, 1060 MPI\_LONG\_LONG\_INT, 34, 227, 702, 861, 1060 MPI\_LOR, 226, 227, 864 MPI\_LXOR, 226, 227, 864 MPI\_MAX, 225-227, 248, 864 MPI\_MAX\_DATAREP\_STRING, 20, 659, 705, 860 MPI\_MAX\_ERROR\_STRING, 20, 469, 475, 860 MPI\_MAX\_INFO\_KEY, 20, 471, 479, 482, 785, 786, 860 MPI\_MAX\_INFO\_VAL, 20, 471, 479, 860 MPI\_MAX\_LIBRARY\_VERSION\_STRING, 20, 452, 860, 1053 MPI\_MAX\_OBJECT\_NAME, 20, 382-385, 860, 1054, 1061 MPI\_MAX\_PORT\_NAME, 21, 535, 860 MPI\_MAX\_PROCESSOR\_NAME, 20, 455, 860, 1062 MPI\_MAX\_PSET\_NAME\_LEN, 506, 860, 1048 MPI\_MAX\_STRINGTAG\_LEN, 344, 361, 860, 1048MPI\_MAXLOC, 226, 229, 230, 233, 864 MPI\_MESSAGE\_NO\_PROC, 89, 91, 92, 101, 859, 1053 MPI\_MESSAGE\_NULL, 89, 91, 92, 864, 1054 MPI\_MIN, 226, 227, 864 "mpi\_minimum\_memory\_alignment", 456, 557, 560, 880, 1047 MPI\_MINLOC, 226, 229, 230, 233, 864 MPI\_MODE\_APPEND, 646, 647, 867

1 MPI\_MODE\_CREATE, 646, 647, 656, 867 MPI\_MODE\_DELETE\_ON\_CLOSE, 646-648,  $\mathbf{2}$ 867 3 MPI\_MODE\_EXCL, 646, 647, 867 4 MPI\_MODE\_NOCHECK, 603, 608, 609, 867 5MPI\_MODE\_NOPRECEDE, 598, 608, 609, 867 6 MPI\_MODE\_NOPUT, 608, 609, 867 7 MPI\_MODE\_NOSTORE, 608, 609, 867 8 MPI\_MODE\_NOSUCCEED, 608, 609, 867 MPI\_MODE\_RDONLY, 646, 647, 652, 867 9 MPI\_MODE\_RDWR, 646, 647, 867 10 MPI\_MODE\_SEQUENTIAL, 646, 647, 650, 657, 11 662, 670, 687, 712, 867, 1063 12MPI\_MODE\_UNIQUE\_OPEN, 646, 647, 867 13 MPI\_MODE\_WRONLY, 646, 647, 867 14MPI\_NO\_OP, 554, 581-583, 864, 1050 15MPI\_NULL\_COPY\_FN, 25, 368, 782, 866 16MPI\_NULL\_DELETE\_FN, 25, 369, 782, 866 MPI\_OFFSET, 35, 227, 702, 861, 862, 171057 - 106018 MPI\_OFFSET\_KIND, 21, 22, 35, 714, 818, 860 19MPI\_OP\_NULL, 237, 864 20MPI\_ORDER\_C, 20, 134, 135, 138, 868 21MPI\_ORDER\_FORTRAN, 20, 134, 135, 138, 22868 23MPI\_PACKED, 17, 33, 34, 45, 46, 175, 177, 182, 701, 702, 854, 861, 862  $^{24}$ MPI\_PROC\_NULL, 32, 36, 38, 85, 89, 91, 92, 25101, 192, 195, 198, 200, 208, 210, 226, 26317, 413, 417, 453, 454, 560, 561, 570, 27859, 1053, 1060, 1062, 1063 28 MPI\_PROD, 226, 227, 864 29MPI\_REAL, 33, 45, 227, 701, 702, 809, 810, 30 816.862 31MPI\_REAL16, <u>35</u>, 227, 703, 862 MPI\_REAL2, 34, 227, 703, 862 32 MPI\_REAL4, 34, 227, 703, 809, 814, 862 33 MPI\_REAL8, 34, 227, 703, 809, 862, 1058 34MPI\_REPLACE, 578, 579, 581, 583, 626, 864, 35 1059.1063 36 MPI\_REQUEST\_NULL, 71-73, 76-80, 636, 864 37 MPI\_ROOT, 192, 859 38 MPI\_SEEK\_CUR, 679, 687, 868 MPI\_SEEK\_END, 679, 687, 868 39 MPI\_SEEK\_SET, 679, 687, 868 40 MPI\_SESSION\_NULL, 502, 864, 1048 41 "mpi\_shared\_memory", 340, 880, 1047 42MPI\_SHORT, 34, 226, 702, 861 43 MPI\_SHORT\_INT, 230, 863 44MPI\_SIGNED\_CHAR, 34, 227, 229, 702, 861, 451060 46MPI\_SIMILAR, 318, 327, 357, 863 "mpi\_size", 503, 507, 511, 880, 1048 47MPI\_SOURCE, 39, 250, 844, 860, 1049, 1055 48

1	MPI_STATUS_IGNORE, 12, 21, <u>41</u> , 42, 635,
2	662, 796, 818, 844, 845, 853, 868, 869,
3	1053
4	MPI_STATUS_SIZE, 21, <u>39</u> , 798, 860, 1055
5	MPI_STATUSES_IGNORE, 20, 21, <u>41</u> , 42, 635,
6	637,818,844,845,868,869
7	$MPI\_SUBARRAYS\_SUPPORTED, 21, 792, 793,$
8	796–798, 800, 804–807, 820, 821, 860,
9	MPI_SUBVERSION, 21, <u>452</u> , 869
10	MPI_SUCCESS, 24, 26, 71, 78, 80, 368, 369,
11	371-374, 376, 377, 394, 469-471, 475-477, 489, 526, 708, 731, 741, 749,
12	752-754, 760, 764, 765, 769, 776, 778,
13	779, 782, 857, 1047
14	MPI_SUM, 226, 227, 578, 853, 864
15	MPI_T_BIND_MPI_COMM, <u>733</u> , 870
16	MPI_T_BIND_MPI_DATATYPE, <u>733</u> , 870
17	MPI_T_BIND_MPI_ERRHANDLER, 733, 870
18	MPI_T_BIND_MPI_FILE, <u>733, 870</u>
19	MPI_T_BIND_MPI_GROUP, 733, 870
20	MPI_T_BIND_MPI_INFO, <u>733</u> , 870
21	MPI_T_BIND_MPI_MESSAGE, 733, 870
	MPI_T_BIND_MPI_OP, <u>733</u> , 870
22	MPI_T_BIND_MPI_REQUEST, 733, 870
23	MPI_T_BIND_MPI_SESSION, <u>733</u> , 870, 1048
24	MPI_T_BIND_MPI_WIN, <u>733</u> , 870
25	MPI_T_BIND_NO_OBJECT, <u>733</u> , 740, 742,
26	749, 751, 764, 765, 870 MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE,
27	760, <u>761</u> , 871
28	MPI_T_CB_REQUIRE_MPI_RESTRICTED,
29	760, 760, 761, 871
30	MPI_T_CB_REQUIRE_NONE, 760, 871
31	MPI_T_CB_REQUIRE_THREAD_SAFE, 760,
32	$\underline{761}, 871$
33	MPI_T_CVAR_HANDLE_NULL, <u>743</u> , 869
34	MPI_T_ENUM_NULL, 739, 749, 764, 869
35	MPI_T_ERR_CANNOT_INIT, 779, 859
36	MPI_T_ERR_CVAR_SET_NEVER, <u>744</u> , 779,
37	859 MPI_T_ERR_CVAR_SET_NOT_NOW, 744,
38	779, 859
39	MPI_T_ERR_INVALID, 779, 859, 1051
40	MPI_T_ERR_INVALID_HANDLE, <u>751</u> , 769,
41	779, 859
42	MPI_T_ERR_INVALID_INDEX, 25, 760, 764,
	$\underline{779}, 786, 859, 1049$
43	MPI_T_ERR_INVALID_ITEM, 25, 786, 859,
44	1049
45	MPI_T_ERR_INVALID_NAME, 741, 749, 765,
46	776, <u>779</u> , 859, 1051
47	MPI_T_ERR_INVALID_SESSION, 779, 859
48	MPI_T_ERR_MEMORY, <u>779</u> , 859

<ul> <li>MPI_T_ERR_NOT_INITIALIZED, 779, 859</li> <li>MPI_T_ERR_NOT_SUPPORTED, 760, 779, 859</li> <li>MPI_T_ERR_OUT_OF_HANDLES, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_ATOMIC, 754, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_STARTSTOP, 752, 753, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_WRITE, 754, 779, 859</li> <li>MPI_T_PVAR_ALL_HANDLES, 752, 753-755, 870</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, 746, 747, 870</li> <li>MPI_T_PVAR_CLASS_COUNTER, 746, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, 747, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, 745, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, 745, 870</li> <li>MPI_T_PVAR_CLASS_NIMER, 746, 870</li> <li>MPI_T_PVAR_CLASS_NIMER, 746, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, 745, 870</li> <li>MPI_T_PVAR_CLASS_NIMER, 746, 870</li> <li>MPI_T_PVAR_CLASS_NIMER, 746, 870</li> <li>MPI_T_PVAR_CLASS_NIMER, 746, 870</li> <li>MPI_T_SCOPE_ALL_FQ, 740, 744, 870</li> <li>MPI_T_SCOPE_ALL_FQ, 740, 870</li> <li>MPI_T_SCOPE_ALL_FQ, 740, 870</li> <li>MPI_T_SCOPE_CONSTANT, 740, 870</li> <li>MPI_T_SCOPE_CONDERED, 759, 871</li> <li>MPI_T_SOURCE_UNORDERED, 759, 871</li> <li>MPI_T_SOURCE_UNORDERED, 759, 871</li> <li>MPI_T_SOURCE_UNORDERED, 759, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, 732, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, 732, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, 732, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, 732, 869<!--</th--><th>MPI_T_ERR_NOT_ACCESSIBLE, <u>761</u>, 762, 779, 859</th></li></ul>	MPI_T_ERR_NOT_ACCESSIBLE, <u>761</u> , 762, 779, 859
<ul> <li>MPI_T_ERR_NOT_SUPPORTED, <u>760</u>, 779, <u>859</u></li> <li>MPI_T_ERR_OUT_OF_HANDLES, <u>779</u>, 859</li> <li>MPI_T_ERR_PVAR_NO_ATOMIC, <u>754</u>, 779, <u>859</u></li> <li>MPI_T_ERR_PVAR_NO_STARTSTOP, <u>752</u>, <u>753</u>, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_WRITE, <u>754</u>, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_WRITE, <u>754</u>, 779, 859</li> <li>MPI_T_PVAR_ALL_HANDLES, <u>752</u>, 753-755, <u>870</u></li> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>746</u>, 747, <u>870</u></li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_TATE, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ONDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_AALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li></ul>	
859 MPI_T_ERR_OUT_OF_HANDLES, 779, 859 MPI_T_ERR_PVAR_NO_ATOMIC, 754, 779, 859 MPI_T_ERR_PVAR_NO_STARTSTOP, 752, 753, 779, 859 MPI_T_ERR_PVAR_NO_WRITE, 754, 779, 859 MPI_T_ERR_PVAR_NO_WRITE, 754, 779, 859 MPI_T_PVAR_ALL_HANDLES, 752, 753-755, 870 MPI_T_PVAR_CLASS_AGGREGATE, 746, 747, 870 MPI_T_PVAR_CLASS_COUNTER, 746, 870 MPI_T_PVAR_CLASS_GENERIC, 747, 870 MPI_T_PVAR_CLASS_GENERIC, 747, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_SIMER, 746, 870 MPI_T_PVAR_CLASS_SIMER, 746, 870 MPI_T_PVAR_CLASS_SIMER, 746, 870 MPI_T_PVAR_CLASS_TIMER, 746, 870 MPI_T_SCOPE_ALL_20, 740, 744, 870 MPI_T_SCOPE_CONSTANT, 740, 870 MPI_T_SCOPE_GROUP_EQ, 740, 744, 870 MPI_T_SCOPE_GROUP_EQ, 740, 744, 870 MPI_T_SCOPE_CONDERD, 759, 871 MPI_T_SOURCE_UNORDERED, 759, 871 MPI_T_SOURCE_UNORDERED, 759, 871 MPI_T_VERBOSITY_MPIDEV_ALL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VER	
<ul> <li>MPI_T_ERR_OUT_OF_SESSIONS, <u>779</u>, 859</li> <li>MPI_T_ERR_PVAR_NO_ATOMIC, <u>754</u>, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_STARTSTOP, <u>752</u>, 753, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_WRITE, <u>754</u>, 779, 859</li> <li>MPI_T_PVAR_ALL_HANDLES, <u>752</u>, 753-755, 870</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>746</u>, 747, 870</li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>746</u>, 870</li> <li>MPI_T_SCOPE_ALL_<u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_<u>80</u>, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP_<u>740</u>, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_UORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ASIC, <u>732</u>, 869</li> <li>MPI_</li></ul>	
<ul> <li>MPI_T_ERR_OUT_OF_SESSIONS, <u>779</u>, 859</li> <li>MPI_T_ERR_PVAR_NO_ATOMIC, <u>754</u>, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_STARTSTOP, <u>752</u>, 753, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_WRITE, <u>754</u>, 779, 859</li> <li>MPI_T_PVAR_ALL_HANDLES, <u>752</u>, 753-755, 870</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>746</u>, 747, 870</li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>746</u>, 870</li> <li>MPI_T_SCOPE_ALL_<u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_<u>80</u>, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP_<u>740</u>, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_UORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ASIC, <u>732</u>, 869</li> <li>MPI_</li></ul>	MPI_T_ERR_OUT_OF_HANDLES, 779, 859
<ul> <li>MPI_T_ERR_PVAR_NO_ATOMIC, <u>754</u>, 779, <u>859</u></li> <li>MPI_T_ERR_PVAR_NO_STARTSTOP, <u>752</u>, <u>753</u>, 779, 859</li> <li>MPI_T_ERR_PVAR_NO_WRITE, <u>754</u>, 779, 859</li> <li>MPI_T_PVAR_ALL_HANDLES, <u>752</u>, 753–755, <u>870</u></li> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>746</u>, 747, <u>870</u></li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>746</u>, 870</li> <li>MPI_T_PVAR_SESSION_NULL, <u>752</u>, 869</li> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_TAG, <u>39</u>, 250, 844, 860, 1049</li></ul>	
859 MPI_T_ERR_PVAR_NO_STARTSTOP, 752, 753, 779, 859 MPI_T_ERR_PVAR_NO_WRITE, 754, 779, 859 MPI_T_PVAR_ALL_HANDLES, 752, 753–755, 870 MPI_T_PVAR_CLASS_AGGREGATE, 746, 747, 870 MPI_T_PVAR_CLASS_COUNTER, 746, 870 MPI_T_PVAR_CLASS_GENERIC, 747, 870 MPI_T_PVAR_CLASS_GENERIC, 747, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_LOWWATERMARK, 746, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_SCOPE_ALL, 740, 870 MPI_T_SCOPE_ALL, 740, 870 MPI_T_SCOPE_ALL, 740, 870 MPI_T_SCOPE_GROUP, 740, 870 MPI_T_SCOPE_GROUP, 740, 870 MPI_T_SCOPE_CONSTANT, 740, 870 MPI_T_SCOPE_CORDERED, 759, 871 MPI_T_SOURCE_ORDERED, 759, 871 MPI_T_SOURCE_ORDERED, 759, 871 MPI_T_VERBOSITY_MPIDEV_ALL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_TAG_39, 250, 844, 860, 1049, 1055 MPI_TAG_9, 36, 453, 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	
753, 779, 859 MPI_T_ERR_PVAR_NO_WRITE, 754, 779, 859 MPI_T_PVAR_ALL_HANDLES, 752, 753-755, 870 MPI_T_PVAR_CLASS_AGGREGATE, 746, 747, 870 MPI_T_PVAR_CLASS_COUNTER, 746, 870 MPI_T_PVAR_CLASS_GENERIC, 747, 870 MPI_T_PVAR_CLASS_HIGHWATERMARK, 746, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_PERCENTAGE, 746, 870 MPI_T_PVAR_CLASS_STATE, 745, 870 MPI_T_PVAR_CLASS_STATE, 745, 870 MPI_T_PVAR_CLASS_STATE, 746, 870 MPI_T_PVAR_CLASS_STATE, 745, 870 MPI_T_PVAR_CLASS_STATE, 746, 870 MPI_T_PVAR_CLASS_TIMER, 746, 870 MPI_T_PVAR_CLASS_TIMER, 746, 870 MPI_T_SCOPE_ALL, 740, 870 MPI_T_SCOPE_ALL, 740, 870 MPI_T_SCOPE_CONSTANT, 740, 870 MPI_T_SOURCE_ORDERED, 759, 871 MPI_T_SOURCE_ORDERED, 759, 871 MPI_T_VERBOSITY_MPIDEV_ALL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_TAG_39, 250, 844, 860, 1049, 1055 MPI_TAG_0B, 36, 453, 849, 852, 863 MPI_THREAD_FUNNELED,	
<ul> <li>MPI_T_PVAR_ALL_HANDLES, <u>752</u>, 753–755, 870</li> <li>MPI_T_PVAR_CLASS_AGGREGATE, <u>746</u>, 747, 870</li> <li>MPI_T_PVAR_CLASS_COUNTER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIND_NULL, <u>752</u>, 869</li> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_CORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_TAG_<u>39</u>, 250, 844, 860, 1049, 1055</li> <li< td=""><td></td></li<></ul>	
870 MPI_T_PVAR_CLASS_AGGREGATE, <u>746</u> , 747, 870 MPI_T_PVAR_CLASS_COUNTER, <u>746</u> , 870 MPI_T_PVAR_CLASS_GENERIC, <u>747</u> , 870 MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>746</u> , 870 MPI_T_PVAR_CLASS_LEVEL, <u>745</u> , 870 MPI_T_PVAR_CLASS_LOWWATERMARK, <u>746</u> , 870 MPI_T_PVAR_CLASS_SIZE, <u>745</u> , 870 MPI_T_PVAR_CLASS_SIZE, <u>745</u> , 870 MPI_T_PVAR_CLASS_STATE, <u>745</u> , 870 MPI_T_PVAR_CLASS_STATE, <u>746</u> , 870 MPI_T_PVAR_CLASS_STATE, <u>746</u> , 870 MPI_T_PVAR_CLASS_TIMER, <u>746</u> , 870 MPI_T_PVAR_CLASS_TIMER, <u>746</u> , 870 MPI_T_PVAR_CLASS_TIMER, <u>746</u> , 870 MPI_T_SCOPE_ALL, <u>740</u> , 870 MPI_T_SCOPE_ALL_EQ, <u>740</u> , 744, 870 MPI_T_SCOPE_GROUP_EQ, <u>740</u> , 744, 870 MPI_T_SCOPE_CONSTANT, <u>740</u> , 870 MPI_T_SCOPE_CONSTANT, <u>740</u> , 870 MPI_T_SCOPE_CONSTENT, <u>740</u> , 870 MPI_T_SCOPE_CONDERED, <u>759</u> , 871 MPI_T_SOURCE_ONDERED, <u>759</u> , 871 MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_ASIC, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI_T_ERR_PVAR_NO_WRITE, <u>754</u> , 779, 859
870 MPI_T_PVAR_CLASS_AGGREGATE, <u>746</u> , 747, 870 MPI_T_PVAR_CLASS_COUNTER, <u>746</u> , 870 MPI_T_PVAR_CLASS_GENERIC, <u>747</u> , 870 MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>746</u> , 870 MPI_T_PVAR_CLASS_LEVEL, <u>745</u> , 870 MPI_T_PVAR_CLASS_LOWWATERMARK, <u>746</u> , 870 MPI_T_PVAR_CLASS_SIZE, <u>745</u> , 870 MPI_T_PVAR_CLASS_SIZE, <u>745</u> , 870 MPI_T_PVAR_CLASS_STATE, <u>745</u> , 870 MPI_T_PVAR_CLASS_STATE, <u>746</u> , 870 MPI_T_PVAR_CLASS_STATE, <u>746</u> , 870 MPI_T_PVAR_CLASS_TIMER, <u>746</u> , 870 MPI_T_PVAR_CLASS_TIMER, <u>746</u> , 870 MPI_T_PVAR_CLASS_TIMER, <u>746</u> , 870 MPI_T_SCOPE_ALL, <u>740</u> , 870 MPI_T_SCOPE_ALL_EQ, <u>740</u> , 744, 870 MPI_T_SCOPE_GROUP_EQ, <u>740</u> , 744, 870 MPI_T_SCOPE_CONSTANT, <u>740</u> , 870 MPI_T_SCOPE_CONSTANT, <u>740</u> , 870 MPI_T_SCOPE_CONSTENT, <u>740</u> , 870 MPI_T_SCOPE_CONDERED, <u>759</u> , 871 MPI_T_SOURCE_ONDERED, <u>759</u> , 871 MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_ASIC, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI_T_PVAR_ALL_HANDLES, <u>752</u> , <u>753</u> – <u>755</u> ,
870 MPI_T_PVAR_CLASS_COUNTER, 746, 870 MPI_T_PVAR_CLASS_GENERIC, 747, 870 MPI_T_PVAR_CLASS_HIGHWATERMARK, 746, 870 MPI_T_PVAR_CLASS_LEVEL, 745, 870 MPI_T_PVAR_CLASS_LOWWATERMARK, 746, 870 MPI_T_PVAR_CLASS_PERCENTAGE, 746, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_STATE, 745, 870 MPI_T_PVAR_CLASS_TIMER, 746, 870 MPI_T_PVAR_CLASS_TIMER, 746, 870 MPI_T_PVAR_LASS_TIMER, 746, 870 MPI_T_PVAR_LASS_TIMER, 746, 870 MPI_T_SCOPE_ALL_EQ, 740, 744, 870 MPI_T_SCOPE_CONSTANT, 740, 870 MPI_T_SCOPE_GROUP_EQ, 740, 744, 870 MPI_T_SCOPE_CONSTANT, 740, 870 MPI_T_SCOPE_CONDERED, 759, 871 MPI_T_SOURCE_ORDERED, 759, 871 MPI_T_SOURCE_UNORDERED, 759, 871 MPI_T_VERBOSITY_MPIDEV_ALL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_TAG, 39, 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, 453, 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	
$\begin{array}{l} MPI_{-}T_{-}VAR_{-}CLASS_{-}COUNTER, \underline{746}, 870\\ MPI_{-}T_{-}VAR_{-}CLASS_{-}HIGHWATERMARK,\\ \underline{746}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}LEVEL, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}LEVEL, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}PERCENTAGE, \underline{746}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}SIZE, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}SIZE, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}SIZE, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}TIMER, \underline{746}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}TIMER, \underline{746}, 870\\ MPI_{-}T_{-}PVAR_{-}LASS_{-}SION_{-}NULL, \underline{752}, 869\\ MPI_{-}T_{-}PVAR_{-}SESSION_{-}NULL, \underline{750}, 869\\ MPI_{-}T_{-}SCOPE_{-}ALL_{-}EQ, \underline{740}, 744, 870\\ MPI_{-}T_{-}SCOPE_{-}GROUP_{-}EQ, \underline{740}, 744, 870\\ MPI_{-}T_{-}SCOPE_{-}GROUP_{-}EQ, \underline{740}, 744, 870\\ MPI_{-}T_{-}SCOPE_{-}CONSTANT, \underline{740}, 870\\ MPI_{-}T_{-}SCOPE_{-}CORDERED, \underline{759}, 871\\ MPI_{-}T_{-}SOURCE_{-}ORDERED, \underline{759}, 871\\ MPI_{-}T_{-}OURCE_{-}ONDERED, \underline{759}, 871\\ MPI_{-}T_{-}VERBOSITY_{-}MPIDEV_{-}DETAIL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}MPIDEV_{-}DETAIL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}TUNER_{-}ALL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}USER_{-}ALL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}USER_{-}AEL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}USER_{-}AEL, \underline{732}, 869\\ MPI_{-}T_{-}VERBO$	
<ul> <li>MPI_T_PVAR_CLASS_GENERIC, <u>747</u>, 870</li> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_CONDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_TAG, <u>39</u>, 250, 844, 860, 1049, 1055</li> <li>MPI_TAE</li></ul>	
<ul> <li>MPI_T_PVAR_CLASS_HIGHWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_LEVEL, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_BESSION_NULL, <u>752</u>, 869</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_TAG, <u>39</u>, 250, 844, 860, 1049, 1055</li> <li>MPI_TAED_FUNNELED, 491, 492, 867</li> </ul>	
$\frac{746}{49}, 870$ MPI_T_PVAR_CLASS_LEVEL, $\frac{745}{45}, 870$ MPI_T_PVAR_CLASS_LOWWATERMARK, $\frac{746}{46}, 870$ MPI_T_PVAR_CLASS_SIZE, $\frac{745}{45}, 870$ MPI_T_PVAR_CLASS_SIZE, $\frac{745}{45}, 870$ MPI_T_PVAR_CLASS_STATE, $\frac{745}{45}, 870$ MPI_T_PVAR_CLASS_TIMER, $\frac{746}{46}, 870$ MPI_T_PVAR_CLASS_TIMER, $\frac{746}{46}, 870$ MPI_T_PVAR_HANDLE_NULL, $\frac{752}{25}, 869$ MPI_T_SCOPE_ALL, $\frac{740}{40}, 870$ MPI_T_SCOPE_ALL_EQ, $\frac{740}{40}, 744, 870$ MPI_T_SCOPE_GROUP_EQ, $\frac{740}{40}, 744, 870$ MPI_T_SCOPE_GROUP_EQ, $\frac{740}{40}, 744, 870$ MPI_T_SCOPE_CONSTANT, $\frac{740}{40}, 870$ MPI_T_SCOPE_CONSTERD, $\frac{759}{59}, 871$ MPI_T_SCOPE_READONLY, $\frac{740}{40}, 870$ MPI_T_SOURCE_ORDERED, $\frac{759}{59}, 871$ MPI_T_SOURCE_UNORDERED, $\frac{759}{59}, 871$ MPI_T_VERBOSITY_MPIDEV_ALL, $\frac{732}{32}, \frac{869}{30}$ MPI_T_VERBOSITY_MPIDEV_DETAIL, $\frac{732}{32}, \frac{869}{30}$ MPI_T_VERBOSITY_TUNER_ALL, $\frac{732}{32}, 869$ MPI_T_VERBOSITY_TUNER_BASIC, $\frac{732}{32}, \frac{869}{30}$ MPI_T_VERBOSITY_USER_ALL, $\frac{732}{32}, 869$ MPI_T_VERBOSITY_USER_ALL, $\frac{732}{32}, 869$ MPI_T_VERBOSITY_USER_ASIC, $\frac{732}{32}, 869$ MPI_T_VERBOSITY_USER_ASIC, $\frac{732}{32}, 869$ MPI_TAG_39, 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, $\frac{453}{3}, 849, 852, 863$ MPI_THREAD_FUNNELED, 491, 492, 867	
$\begin{array}{l} MPI_{-}PVAR_{-}CLASS_{-}LEVEL, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}LOWWATERMARK,\\ \underline{746}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}SIZE, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}SIZE, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}STATE, \underline{745}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}TIMER, \underline{746}, 870\\ MPI_{-}T_{-}PVAR_{-}CLASS_{-}TIMER, \underline{746}, 870\\ MPI_{-}T_{-}PVAR_{-}LASS_{-}SIMER, \underline{740}, 870\\ MPI_{-}T_{-}SCOPE_{-}ALL_{-}EQ, \underline{740}, 744, 870\\ MPI_{-}T_{-}SCOPE_{-}GROUP_{-}EQ, \underline{740}, 744, 870\\ MPI_{-}T_{-}SCOPE_{-}GROUP_{-}EQ, \underline{740}, 744, 870\\ MPI_{-}T_{-}SCOPE_{-}GROUP_{-}EQ, \underline{740}, 744, 870\\ MPI_{-}T_{-}SCOPE_{-}CONSTANT, \underline{740}, 870\\ MPI_{-}T_{-}SOURCE_{-}ORDERED, \underline{759}, 871\\ MPI_{-}T_{-}SOURCE_{-}ORDERED, \underline{759}, 871\\ MPI_{-}T_{-}SOURCE_{-}ORDERED, \underline{759}, 871\\ MPI_{-}T_{-}VERBOSITY_{-}MPIDEV_{-}ALL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}MPIDEV_{-}DETAIL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}TUNER_{-}DETAIL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}USER_{-}ALL, \underline{732}, 869\\ MPI_{-}T_{-}VERBOSITY_{-}USE$	
<ul> <li>MPI_T_PVAR_CLASS_LOWWATERMARK, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_BESSION_NULL, <u>750</u>, 869</li> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_TAG, <u>39</u>, 250, 844, 860, 1049, 1055</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	
$\frac{746}{870}$ MPI_T_PVAR_CLASS_PERCENTAGE, 746, 870 MPI_T_PVAR_CLASS_SIZE, 745, 870 MPI_T_PVAR_CLASS_STATE, 745, 870 MPI_T_PVAR_CLASS_TIMER, 746, 870 MPI_T_PVAR_CLASS_TIMER, 746, 870 MPI_T_PVAR_SESSION_NULL, 750, 869 MPI_T_SCOPE_ALL, 740, 870 MPI_T_SCOPE_ALL_EQ, 740, 744, 870 MPI_T_SCOPE_GROUP, 740, 870 MPI_T_SCOPE_GROUP_EQ, 740, 744, 870 MPI_T_SCOPE_GROUP_EQ, 740, 744, 870 MPI_T_SCOPE_CONSTANT, 740, 870 MPI_T_SCOPE_CONDERED, 759, 871 MPI_T_SOURCE_ORDERED, 759, 871 MPI_T_SOURCE_UNORDERED, 759, 871 MPI_T_VERBOSITY_MPIDEV_ALL, 732, 869 MPI_T_VERBOSITY_MPIDEV_DETAIL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_ALL, 732, 869 MPI_T_VERBOSITY_TUNER_DETAIL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ALL, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_T_VERBOSITY_USER_ASIC, 732, 869 MPI_TAG_ 0B, 36, 453, 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867 	
<ul> <li>MPI_T_PVAR_CLASS_PERCENTAGE, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_SESSION_NULL, <u>750</u>, 869</li> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_TAG, <u>39</u>, 250, 844, 860, 1049, 1055</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	
<ul> <li>MPI_T_PVAR_CLASS_SIZE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_BADLE_NULL, <u>752</u>, 869</li> <li>MPI_T_PVAR_SESSION_NULL, <u>750</u>, 869</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	
<ul> <li>MPI_T_PVAR_CLASS_STATE, <u>745</u>, 870</li> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_HANDLE_NULL, <u>752</u>, 869</li> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPL T PVAR CLASS SIZE $745$ 870
<ul> <li>MPI_T_PVAR_CLASS_TIMER, <u>746</u>, 870</li> <li>MPI_T_PVAR_HANDLE_NULL, <u>752</u>, 869</li> <li>MPI_T_PVAR_SESSION_NULL, <u>750</u>, 869</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPLT PVAR CLASS STATE $745$ , 870
<ul> <li>MPI_T_PVAR_HANDLE_NULL, <u>752</u>, 869</li> <li>MPI_T_PVAR_SESSION_NULL, <u>750</u>, 869</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	
<ul> <li>MPI_T_PVAR_SESSION_NULL, <u>750</u>, 869</li> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_BRADONLY, <u>740</u>, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	
<ul> <li>MPI_T_SCOPE_ALL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPI_T_PVAR_HANDLE_NOLL, $\underline{152}$ , 809 MDI_T_DVAR_SESSION_NULL_750_860
<ul> <li>MPI_T_SCOPE_ALL_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPL T SCOPE ALL 740 870
<ul> <li>MPI_T_SCOPE_CONSTANT, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MDL T SCOPE ALL EO 740 744 870
<ul> <li>MPI_T_SCOPE_GROUP, <u>740</u>, 870</li> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPI_I_SCOPE_ALL_EQ, <u>140</u> , 144, 870
<ul> <li>MPI_T_SCOPE_GROUP_EQ, <u>740</u>, 744, 870</li> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_UNER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_UNER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	$MPI_TSCOPE_CONSTANT, \underline{740}, 870$
<ul> <li>MPI_T_SCOPE_LOCAL, <u>740</u>, 870</li> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPI_I_SCOPE_GROUP, <u>740</u> , 870
<ul> <li>MPI_T_SCOPE_READONLY, <u>740</u>, 870</li> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPI_T_SCOPE_GROUP_EQ, <u>740</u> , 744, 870
<ul> <li>MPI_T_SOURCE_ORDERED, <u>759</u>, 871</li> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPI_I_SCOPE_LOCAL, <u>740</u> , 870
<ul> <li>MPI_T_SOURCE_UNORDERED, <u>759</u>, 871</li> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	MPI_I_SCOPE_READONLY, <u>740</u> , 870
<ul> <li>MPI_T_VERBOSITY_MPIDEV_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_ALL, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_BASIC, <u>732</u>, 869</li> <li>MPI_T_VERBOSITY_USER_DETAIL, <u>732</u>, 869</li> <li>MPI_TAG_ <u>39</u>, 250, 844, 860, 1049, 1055</li> <li>MPI_TAG_UB, 36, <u>453</u>, 849, 852, 863</li> <li>MPI_THREAD_FUNNELED, 491, 492, 867</li> </ul>	
MPI_T_VERBOSITY_MPIDEV_BASIC, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_USER_ALL, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , <u>869</u> MPI_TAG, <u>39</u> , <u>250</u> , <u>844</u> , <u>860</u> , 1049, 1055 MPI_TAG_UB, <u>36</u> , <u>453</u> , <u>849</u> , <u>852</u> , <u>863</u> MPI_THREAD_FUNNELED, <u>491</u> , <u>492</u> , <u>867</u>	MPI_I_SOURCE_UNORDERED, <u>759</u> , 871
869 MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	
MPI_T_VERBOSITY_MPIDEV_DETAIL, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u> , <u>869</u> MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	
869 MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_TUNER_DETAIL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	
869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	869
869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI_T_VERBOSITY_TUNER_ALL, <u>732</u> , 869
869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI_T_VERBOSITY_TUNER_BASIC, <u>732</u> , 869
869 MPI_T_VERBOSITY_USER_ALL, <u>732</u> , 869 MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI_T_VERBOSITY_TUNER_DETAIL, 732,
MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, <u>36</u> , <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, <u>491</u> , 492, 867	869
MPI_T_VERBOSITY_USER_BASIC, <u>732</u> , 869 MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, <u>36</u> , <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, <u>491</u> , 492, 867	MPI_T_VERBOSITY_USER_ALL, 732, 869
MPI_T_VERBOSITY_USER_DETAIL, <u>732</u> , 869 MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI_T_VERBOSITY_USER_BASIC, 732, 869
MPI_TAG, <u>39</u> , 250, 844, 860, 1049, 1055 MPI_TAG_UB, <u>36</u> , <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI_T_VERBOSITY_USER_DETAIL. 732, 869
MPI_TAG_UB, 36, <u>453</u> , 849, 852, 863 MPI_THREAD_FUNNELED, 491, 492, 867	MPI TAG. 39, 250, 844, 860, 1049, 1055
MPI_THREAD_FUNNELED, 491, 492, 867	
	"MPI_THREAD_FUNNELED", 501, 880

MPI\_THREAD\_MULTIPLE, 491, 492, 494, 867, 1051 "MPI\_THREAD\_MULTIPLE", 501, 880 MPI\_THREAD\_SERIALIZED, 491, 492, 867 "MPI\_THREAD\_SERIALIZED", 501, 880 MPI\_THREAD\_SINGLE, 491-493, 501, 502, 867 "MPI\_THREAD\_SINGLE", 501, 880 MPI\_TYPE\_DUP\_FN, <u>376</u>, 865 MPI\_TYPE\_NULL\_COPY\_FN, 376, 865 MPI\_TYPE\_NULL\_DELETE\_FN, 376, 865, 1056 MPI\_TYPECLASS\_COMPLEX, 815, 868 MPI\_TYPECLASS\_INTEGER, 815, 868 MPI\_TYPECLASS\_REAL, 815, 868 MPI\_UB, 4, 25, 788, 1052 MPI\_UINT16\_T, <u>34</u>, 227, 702, 861, 1057-1060 MPI\_UINT32\_T, <u>34</u>, 227, 702, 736, 861, 1057 - 1060MPI\_UINT64\_T, <u>34</u>, 227, 702, 736, 861, 1057 - 1060MPI\_UINT8\_T, 34, 227, 702, 861, 1057-1060 MPI\_UNDEFINED, 23, 40, 41, 59, 76, 77, 80, 81, 144, 148, 150, 155, 179, 316, 317, 335, 336, 339, 403, 415, 416, 701, 706, 811, 812, 859, 1053, 1060 MPI\_UNEQUAL, 318, 327, 357, 863 MPI\_UNIVERSE\_SIZE, 522, 544, 866 MPI\_UNSIGNED, <u>34</u>, <u>226</u>, <u>702</u>, <u>736</u>, <u>745–747</u>, 861 MPI\_UNSIGNED\_CHAR, 34, 227, 229, 702, 861 MPI\_UNSIGNED\_LONG, <u>34</u>, 227, 702, 736, 745-747,861 MPI\_UNSIGNED\_LONG\_LONG, 34, 227, 702, 736, 745–747, 861, 1060 MPI\_UNSIGNED\_SHORT, <u>34</u>, 226, 702, 861 MPI\_UNWEIGHTED, 21, 398, 400-402, 410, 411, 818, 868, 1053, 1059 MPI\_VAL, 18, 842 MPI\_VERSION, 21, 452, 869 MPI\_WCHAR, 34, 229, 385, 701, 702, 861, 1060 MPI\_WEIGHTS\_EMPTY, 21, 398, 400, 818, 868, 1053 MPI\_WIN\_BASE, 566, 852, 866 MPI\_WIN\_CREATE\_FLAVOR, 566, 866 MPI\_WIN\_DISP\_UNIT, 566, 866 MPI\_WIN\_DUP\_FN, <u>373</u>, 865 MPI\_WIN\_FLAVOR\_ALLOCATE, 567, 866 MPI\_WIN\_FLAVOR\_CREATE, 567, 866 MPI\_WIN\_FLAVOR\_DYNAMIC, 567, 866 MPI\_WIN\_FLAVOR\_SHARED, 561, 567, 866 MPI\_WIN\_MODEL, 566, 593, 866 MPI\_WIN\_NULL, 565, 864 MPI\_WIN\_NULL\_COPY\_FN, 373, 865 MPI\_WIN\_NULL\_DELETE\_FN, 373, 865

MPI_WIN_SEPARATE, 567, 592, 593, 612, 866	1
MPI_WIN_SIZE, <u>566</u> , 866	2
MPI_WIN_UNIFIED, 567, <u>593</u> , 612, 621, 866	3
MPI_WTIME_IS_GLOBAL, <u>453</u> , 454, 478, 849,	4
863 "mpix://UNIVERSE", 503	5
	6
"native", 698–700, 709, 879	7
"nb_proc", 656, 880	8
"no_locks", 554, 566, 880	9
"none", 619, 880	10
"num_io_nodes", 656, 880	11
"path", 531, 880	12
path , 001, 000	13
"random", 655, 880	14
"rar", 619, 880	15
"raw", 619, 880	16
"read_mostly", $655$ , $881$	17
"read_once", 655, 881	18
"reverse_sequential", 655, 881	19
"same_disp_unit", 554, 880, 1051	20
"same_op", 554, 881	
"same_op_no_op", 554, 881	21
"same_size", 554, 880, 1050	22
"sequential", 655, 881	23
"soft", 490, 526, 531, 880	24
"striping_factor", 656, 880	25
"striping_unit", 656, 880	26
"thread_level", 490, 501, 880	27
"true", 346, 566, 655, 881	28
the , 510, 500, 500, 001	29
"war", 619, 881	30
"waw", 619, 881	31
"wdir", 490, 530, 880	32
"write_mostly", 655, 881	33
"write_once", 655, 881	34
	35
	36
	37
	38
	39
	40
	41
	42
	43
	44
	44 45
	46
	47
	48

## **MPI Declarations Index**

10

 $11 \\ 12$ 

This index refers to declarations needed in C and Fortran, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas). Fortran defined types are shown as TYPE(*name*).

13	MPI_Aint, <u>22</u> , 22, 35, <u>121</u> , 124, 127, 131,
14	141-143, 147-149, 159, 182-184, 553,
15	556, 558, 562, 563, 566, 570, 573, 577,
16	579, 582 - 584, 586, 587, 589, 700, 705,
17	759, 818, 849, 861, 871
18	MPI_Comm, 18, <u>32</u> , 318, 325–330, <u>333</u> , 335,
	$338, 345-347, 357, 358, \underline{359}, 360, 361,$
19	367, 370, 371, 505, 863, 864, 871
20	TYPE(MPI_Comm), <u>32</u> , <u>333</u> , <u>359</u> , 805, 863,
21	864, 872
22	MPI_Count, <u>22</u> , 23, 35, 758, 759, 861, 871, 1053
23	MPI_Datatype, <u>121</u> , 376, 377, 827, 861–864, 871
24	TYPE(MPI_Datatype), <u>121</u> , 861–864, 872
25	MPI_Errhandler, <u>461</u> , 462–468, 842, 860, 864,
26	871 TVDE(MDI Errhandlar) 461 860 864 879
27	TYPE(MPI_Errhandler), <u>461</u> , 860, 864, 872 MPI_F08_status, 844, 869, 871, 1056
28	MPI_File, 465, 466, <u>645</u> , 648–651, 653, 654,
	$\begin{array}{c} 656, \ 658, \ 662-668, \ 670-674, \ 676-683, \end{array}$
29	685-687, 689-696, 700, 710-712, 842,
30	864, 871
31	TYPE(MPI_File), <u>645</u> , 864, 872
32	MPI_Fint, 841, 869, 871, 1060
33	MPI_Group, <u>316</u> , 317–322, 324, 358, 505, 567,
34	598, 599, 651, 842, 864, 871
35	TYPE(MPI_Group), <u>316</u> , 864, 872
36	MPI_Info, 455, 479, 480-483, 503, 507, 522,
37	526, 527, 535, 538–540, 548, 568, 645,
38	648,653,654,656,784,842,863,864,
39	871,1062
40	TYPE(MPI_Info), <u>479</u> , 863, 864, 872
	MPI_Message, <u>87</u> , 842, 859, 864, 871, 1054
41	TYPE(MPI_Message), <u>87</u> , 859, 864, 872
42	$MPI\_Offset, \underline{22}, 22, 35, 649651, 656, 658,$
43	662-668, 678-680, 687, 689, 691, 706,
44	<u>714</u> , 759, 841, 861, 871
45	MPI_Op, 224, <u>233</u> , 237, 238, 240–242, 244,
46	246-248, 269-271, 273-275, 294-296,
47	298–300, 577, 579, 582, 587, 589, 842,
48	864, 871
-	

TYPE(MPI\_Op), 233, 864, 872 MPI\_Request, 62-67, 69, 71, 72, 73, 76-79, 81, 83, 92-100, 106-110, 470, 634, 637, 666-668, 674, 676, 677, 682, 683, 820, 842, 864, 871 TYPE(MPI\_Request), 71, 864, 872 MPI\_Session, 467, 468, 501, 842, 864, 871, 1048 TYPE(MPI\_Session), 501, 864, 872, 1048 MPI\_Status, 36, 39, 41-43, 71, 72, 76-79, 81, 83-85, 87, 89, 90, 93, 154, 155, 635, 640, 641, 662–665, 670–673, 681, 685, 686, 690, 692–696, 795, 843–846, 868, 871, 1049, 1053, 1055 TYPE(MPI\_Status), <u>36</u>, 39, 805, 844-846, 868, 872, 1045, 1049, 1055 MPI\_T\_cb\_safety, 760, 767, 768, 871, 1048 MPI\_T\_cvar\_handle, 742, 743, 869, 871 MPI\_T\_enum, 736, 737, 738, 747, 762, 869, 871 MPI\_T\_event\_instance, 767, 771-773, 871, 1048 MPI\_T\_event\_registration, 765, 766-768, 770, 871.1048 MPI\_T\_pvar\_handle, 750, 751-754, 869-871 MPI\_T\_pvar\_session, 750, 751-754, 869, 871 MPI\_T\_source\_order, 758, 759, 871, 1048 MPI\_Win, 373-375, 463, 464, 553, 556, 558, 562, 565, 567, 568, 570, 573, 577, 579, 582-584, 586, 587, 589, 596, 598-600, 602-607, 842, 864, 871 TYPE(MPI\_Win), <u>553</u>, <u>556</u>, <u>558</u>, <u>562</u>, 864, 872

## MPI Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. Fortran example prototypes are given near the text of the C name.

COMM_COPY_ATTR_FUNCTION, 24, 25, 367, 368, 794, 865, 877	MF
COMM_DELETE_ATTR_FUNCTION, 25, 367,	M
<u>368,</u> 865, 877 COMM_ERRHANDLER_FUNCTION, <u>461,</u> 877	MF
COPY_FUNCTION, 25, 781, <u>782</u> , 866, 879	M
DATAREP_CONVERSION_FUNCTION, 705,	M
$\underline{707}, 865, 878$	MF
DATAREP_EXTENT_FUNCTION, 705, 706, 878	MF
DELETE_FUNCTION, 25, 781, 782, 866, 879	M
FILE_ERRHANDLER_FUNCTION, 465, 878	M
GREQUEST_CANCEL_FUNCTION, 634, 636, 878	M
GREQUEST_FREE_FUNCTION, 634, 635, 878	M
GREQUEST_QUERY_FUNCTION, 634, 635, 878	MF
MPI_Comm_copy_attr_function, 24, 25, 367,	MF
$367, \underline{368}, 794, 865, 872, 874$	
MPI_Comm_delete_attr_function, 25, <u>367</u> , 367,	M
<u>368, 865, 872, 874</u>	MF
MPI_Comm_errhandler_fn, <u>784</u> , <u>1059</u> MPI_Comm_errhandler_function, <u>25</u> , <u>461</u> , <u>784</u> ,	M
788, 873, 875, 1059	
MPI_Copy_function, 25, <u>781</u> , 866, 878	<u>с</u> г
MPI_Datarep_conversion_function, 704, 706,	SE
839, 865, 873, 876	
MPI_Datarep_conversion_function_c, 704, <u>706</u> ,	ΤY
839, 865, 873, 876, 1046 MPI_Datarep_extent_function, 704, 705, 706,	
873, 876	ΤY
MPI_Delete_function, 25, 781, <u>782</u> , 866, 878	
MPI_File_errhandler_fn, 784, 1059	US
MPI_File_errhandler_function, <u>465</u> , 784, 873,	
875,1059	WI

$MPI_Grequest\_cancel\_function, 634, \underline{636, 873,}$	15
876	16
MPI_Grequest_free_function, 634, <u>635</u> , 873, 876	17
MPI_Grequest_query_function, 634, <u>635</u> , 873, 875	18
MPI_Handler_function, 25, <u>788</u> , 1052	19
MPI_Session_errhandler_function, 466, 467, 873,	20
875, 1048	21
MPI_T_event_cb_function, <u>767</u> , 873, 1048	22
$MPI_T\_event\_dropped\_cb\_function, \underline{770}, 873,$	23
1048	$^{24}$
MPI_T_event_free_cb_function, <u>769</u> , 873, 1048	25
MPI_Type_copy_attr_function, <u>376</u> , <u>376</u> , <u>377</u> , 865, 872, 875	26
MPI_Type_delete_attr_function, <u>376</u> , <u>377</u> , 865,	27
873, 875, 1056	28
MPI_User_function, 233, 234, 238, 839, 872, 874	29
MPI_User_function_c, 233, 234, 839, 872, 874,	30
1046	31
MPI_Win_copy_attr_function, 372, <u>373</u> , 865,	32
872, 874	33
MPI_Win_delete_attr_function, 372, <u>373</u> , 865, 872, 874	34
MPI_Win_errhandler_fn, $\frac{784}{1059}$	35
MPI_Win_errhandler_function, <u>463</u> , 784, 873,	36
875, 1059	37
	38
SESSION_ERRHANDLER_FUNCTION, <u>467</u> ,	39
878,1048	40
TYPE_COPY_ATTR_FUNCTION, <u>376</u> , <u>377</u> ,	41
865, 877	42
TYPE_DELETE_ATTR_FUNCTION, 376, <u>377</u> ,	43
865,877	44
USER_FUNCTION, 233, <u>234</u> , 876	45
OSEN_1 ONCTION, 233, <u>234</u> , 010	46
WIN_COPY_ATTR_FUNCTION, <u>373</u> , <u>374</u> , <u>865</u> ,	47
877	48

- 1	090	

1	WIN_DELETE_ATTR_FUNCTION, 373, <u>374</u> ,
2	865, 877
3	WIN_ERRHANDLER_FUNCTION, <u>463</u> , 878
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	

### **MPI** Function Index

The underlined page numbers refer to the function definitions.

MPI\_ABORT, 235, 459, 489, 495, 514, 546, 735, 841, 1062 MPI\_ACCUMULATE, 551, 569, 577, 578, 582, 589, 593, 619, 625, 626, 1059, 1062, 1063 MPI\_Accumulate\_c, 577, 1046 MPI\_ADD\_ERROR\_CLASS, 473 MPI\_ADD\_ERROR\_CODE, 474  $\mathsf{MPI}\_\mathsf{ADD}\_\mathsf{ERROR}\_\mathsf{STRING},\,\underline{474},\,475$ MPI\_ADDRESS, 25, 787, 804, 1052 MPI\_AINT\_ADD, 26, 141, 142, 563, 1051 MPI\_AINT\_DIFF, 26, 141, 142, 143, 563, 1051 MPI\_ALLGATHER, 187, 191, 192, 213, 214, 216, 218, 261 MPI\_Allgather\_c, 213, 1046 MPI\_ALLGATHER\_INIT, 187, 191, 192, 286, 1046 MPI\_Allgather\_init\_c, 286, 1046 MPI\_ALLGATHERV, 187, 191, 192, 215, 216, 263MPI\_Allgatherv\_c,  $\underline{215}$ , 1046MPI\_ALLGATHERV\_INIT, 187, 191, 192, 287, 1046MPI\_Allgatherv\_init\_c, 287, 1046 MPI\_ALLOC\_MEM, 455, 456-458, 471, 555, 557, 559-561, 564, 572, 605, 804-806, 818, 1047, 1049, 1056 MPI\_ALLOC\_MEM\_CPTR, 456, 1049 MPI\_ALLREDUCE, 187, 190-192, 226, 234, 239, 271, 1060 MPI\_Allreduce\_c, <u>239</u>, 1046 MPI\_ALLREDUCE\_INIT, 187, 191, 192, 295, 1046 MPI\_Allreduce\_init\_c, 295, 1046 MPI\_ALLTOALL, 187, 191, 192, 217, 218, 220, 221, 264, 1058 MPI\_Alltoall\_c, 217, 1046 MPI\_ALLTOALL\_INIT, 187, 191, 192, 289, 1046 MPI\_Alltoall\_init\_c, <u>289</u>, 1046 MPI\_ALLTOALLV, 187, 191, 192, 219, 220, 221, 223, 266, 1058 MPI\_Alltoallv\_c, <u>219</u>, 1046

MPI_ALLTOALLV_INIT, 187, 191, 192, <u>290</u> ,	11
1046	12
MPI_Alltoallv_init_c, <u>290</u> , 1046	13
MPI_ALLTOALLW, 187, 191, 192, <u>221</u> , 222,	14
223, 268, 1058	15
MPI_Alltoallw_c, <u>222</u> , 1046	16
MPI_ALLTOALLW_INIT, 187, 191, 192, <u>292</u> ,	
1046	17
MPI_Alltoallw_init_c, <u>292</u> , <u>1046</u>	18
MPI_ATTR_DELETE, 25, 379, 782, <u>784</u>	19
MPI_ATTR_GET, 25, 379, <u>783</u> , 849	20
MPI_ATTR_PUT, 25, 379, <u>783</u> , 849, 852, 853	21
MPI_BARRIER, 187, 191, 192, <u>194</u> , 252, 513,	22
615-617, 716	23
MPI_BARRIER_INIT, 187, 191, 192, <u>277</u> , 1046	24
MPI_BCAST, 16, 187, 191, 192, <u>194</u> , 195, 225,	25
253, 305	26
MPI_Bcast_c, <u>194</u> , 1046	
MPI_BCAST_INIT, 16, 187, 191, 192, <u>278</u> , 1046	27
MPI_Bcast_init_c, <u>278</u> , 1046	28
MPI_BSEND, 16, <u>50</u> , 51, 60	29
MPI_Bsend_c, <u>50</u> , 1046	30
MPI_BSEND_INIT, <u>96</u> , 100	31
MPI_Bsend_init_c, <u>96</u> , 1046	32
MPI_BUFFER_ATTACH, <u>57</u> , 72	33
MPI_Buffer_attach_c, <u>58</u> , <u>1046</u> MPI_BUFFER_DETACH, <u>58</u> , <u>1046</u> , <u>1056</u>	34
MPI_Buffer_detach_c, <u>58</u> , 1046, 1050	35
MPI_CANCEL, 25, 54, 72, 84, <u>92</u> , 93, 94, 251,	
$\begin{array}{c} 277, 496, 584, 633, 636, 637, 784, 1046 \end{array}$	36
MPI_CART_COORDS, 391, <u>407</u> , 1061	37
MPI_CART_CREATE, 356, 391, <u>392</u> , 393–395,	38
415, 417, 819, 1049, 1061	39
MPI_CART_GET, 391, <u>405</u> , 1061	40
MPI_CART_MAP, 391, <u>415</u> , 1054	41
MPI_CART_RANK, 391, <u>406</u> , 1061	42
MPI_CART_SHIFT, 391, <u>412</u> , 413, 417, 1061	43
MPI_CART_SUB, 391, <u>413</u> , 414, 415, 1062	44
MPI_CARTDIM_GET, 391, <u>405</u> , 1061	
MPI_CLOSE_PORT, <u>536</u> , 539	45
MPI_COMM_ACCEPT, 534, 535, <u>536</u> , 537, 538,	46
544, 546	47
,	48

1	MPI_COMM_C2F, 842
2	MPI_COMM_CALL_ERRHANDLER, <u>475</u> , 477
3	MPI_COMM_COMPARE, 327, 357
5	
4	MPI_COMM_CONNECT, 472, <u>537</u> , 538, 544,
5	546
6	MPI_COMM_CREATE, 325, 327, <u>331</u> , 332–336,
	391,1058
7	MPI_COMM_CREATE_ERRHANDLER, 25, 460,
8	461, 462, 787, 875, 877, 1056
9	MPI_COMM_CREATE_FROM_GROUP, 325,
10	327, 343, 344, 361, 453, 1048
11	MPI_COMM_CREATE_GROUP, 325, 327, <u>334</u> ,
12	335, 336, 344, 1054
13	MPI_COMM_CREATE_KEYVAL, 25, 365, <u>367</u> ,
	368, 369, 379, 781, 848, 849, 874, 877,
14	1056, 1060
15	MPI_COMM_DELETE_ATTR, 25, 365, 369,
16	370, <u>372</u> , 379, 783
17	MPI_COMM_DISCONNECT, 379, 502, 503,
	513, 527, 546, <u>547</u>
18	
19	MPI_COMM_DUP, 318, 325, 327, <u>328</u> , 329,
20	330, 332, 359, 362, 366, 368, 372, 379,
21	387, 781, 789, 1047, 1054
	MPI_COMM_DUP_FN, 25, <u>368</u> , 369, 799, 865,
22	1051, 1056
23	MPI_COMM_DUP_WITH_INFO, 325, 327, 329,
24	$330, 345, 366, 368, 372, 379, 789, \overline{1054}$
25	MPI_COMM_F2C, 842
	MPI_COMM_FREE, 325, 328, <u>345</u> , 359, 362,
26	
27	369, 370, 372, 379, 495, 499, 502, 513,
28	527, 546, 547, 782
29	MPI_COMM_FREE_KEYVAL, 25, 365, <u>370</u> ,
	379, 782
30	MPI_COMM_GET_ATTR, 25, 365, 370, <u>371</u> ,
31	379, 453, 783, 800, 849, 852
32	MPI_COMM_GET_ERRHANDLER, 25, 460,
33	462,787,1062
	MPI_COMM_GET_INFO, 345, <u>347</u> , 1047, 1054
34	MPI_COMM_GET_NAME, 382, 383, 384, 1061
35	MPI_COMM_GET_PARENT, 383, 489, 490,
36	
37	495, 500, 524, <u>527</u>
	MPI_COMM_GROUP, 19, 316, <u>318</u> , 319, 325,
38	326,357,460,1062
39	MPI_COMM_IDUP, 325, 327, <u>329</u> , 330, 355,
40	366, 368, 372, 379, 789, 1047, 1049,
41	1054
	MPI_COMM_IDUP_WITH_INFO, 325, 327,
42	330, 345, 366, 368, 372, 379, 789, 1047
43	MPI_COMM_JOIN, 500, 548, 549
44	
45	MPI_COMM_NULL_COPY_FN, 24, 25, <u>368</u> ,
	369, 794, 848, 865, 1051, 1056
46	MPI_COMM_NULL_DELETE_FN, 25, <u>369</u> , 865,
47	1051
48	MPI_COMM_RANK, 16, <u>326</u> , 357, 802
	/ /

- MPI\_COMM\_RANK\_F08, 802
- MPI\_COMM\_REMOTE\_GROUP, 358
- MPI\_COMM\_REMOTE\_SIZE, 358
- MPI\_COMM\_SET\_ATTR, 25, 365, 369, <u>370</u>, 379, 782, 800, 849, 853
- MPI\_COMM\_SET\_ERRHANDLER, 25, 460, <u>462</u>, 787
- MPI\_COMM\_SET\_INFO, 277, 345, <u>346</u>, 347, 1047, 1049, 1054
- MPI\_COMM\_SET\_NAME, <u>382</u>
- MPI\_COMM\_SIZE, <u>325</u>, 326, 357, 624
- MPI\_COMM\_SPAWN, 489, 490, 515, 516, 521, 522, 523, 524–527, 529–531, 544, 545, 1048
- MPI\_COMM\_SPAWN\_MULTIPLE, 489, 490, 516, 521, 522, 527, 528, 529, 545, 1048
- MPI\_COMM\_SPLIT, 325, 327, 331, 332, <u>335</u>, 336, 337, 387, 391, 393, 395, 414–416, 1058
- MPI\_COMM\_SPLIT\_TYPE, 325, 327, <u>339</u>, 342, 345, 1046, 1047, 1054
- MPI\_COMM\_TEST\_INTER, 356, 357
- MPI\_COMPARE\_AND\_SWAP, 551, 569, <u>583</u>, 624
- MPI\_CONVERSION\_FN\_NULL, <u>708</u>, 865, 1050 MPI\_CONVERSION\_FN\_NULL\_C, <u>708</u>, 839,
- 865, 1046
- MPI\_DIMS\_CREATE, 391, <u>393</u>, 394, 1047
- MPI\_DIST\_GRAPH\_CREATE, 345, 391, 396, <u>399</u>, 400–402, 411, 412, 417, 1059
- MPI\_DIST\_GRAPH\_CREATE\_ADJACENT, 345, 391, 396, <u>397</u>, 398, 402, 411, 416, 1055, 1059
- MPI\_DIST\_GRAPH\_NEIGHBORS, 391, 410, 411, 417, 1055, 1059
- MPI\_DIST\_GRAPH\_NEIGHBORS\_COUNT, 391, <u>410</u>, 411, 412, 1052, 1059
- MPI\_DUP\_FN, 25, 368, 782, 866
- MPI\_ERRHANDLER\_C2F, 513, 842
- MPI\_ERRHANDLER\_CREATE, 25, <u>787</u>, 1052, 1056
- MPI\_ERRHANDLER\_F2C, 513, 842
- MPI\_ERRHANDLER\_FREE, 460, <u>468</u>, 495, 502, 513, 1062
- MPI\_ERRHANDLER\_GET, 25, 787, 1052, 1062
- MPI\_ERRHANDLER\_SET, 25, <u>787</u>, 1052
- MPI\_ERROR\_CLASS, <u>470</u>, 473, 513
- MPI\_ERROR\_STRING, 469, 470, 473, 475, 513
- MPI\_EXSCAN, 188, 191, 226, 234, <u>247</u>, 248, 276, 1058
- MPI\_Exscan\_c, 247, 1046
- MPI\_EXSCAN\_INIT, 188, 191, 300, 1046
- MPI\_Exscan\_init\_c, <u>300</u>, 1046
- MPI\_F\_SYNC\_REG, 142, 626, 792, 808, 809,

828, 830-833, 1057 MPI\_FETCH\_AND\_OP, 551, 569, 578, 582 MPI\_FILE\_C2F, 842 MPI\_FILE\_CALL\_ERRHANDLER, 476, 477 MPI\_FILE\_CLOSE, 547, 645, 646, 648 MPI\_FILE\_CREATE\_ERRHANDLER, 460, 465, 466, 875, 878, 1056 MPI\_FILE\_DELETE, 647, 648, 649, 653, 655, 720 MPI\_FILE\_F2C, 842 MPI\_FILE\_GET\_AMODE, 652 MPI\_FILE\_GET\_ATOMICITY, 711, 712 MPI\_FILE\_GET\_BYTE\_OFFSET, 670, 679, <u>680, 688</u> MPI\_FILE\_GET\_ERRHANDLER, 460, 466, 720, 1062MPI\_FILE\_GET\_GROUP, 651 MPI\_FILE\_GET\_INFO, 653, 654, 655, 1047, 1063 MPI\_FILE\_GET\_POSITION, 679 MPI\_FILE\_GET\_POSITION\_SHARED, 687, 688, 712 MPI\_FILE\_GET\_SIZE, 651, 715 MPI\_FILE\_GET\_TYPE\_EXTENT, 699, 700, 708, 1046 MPI\_File\_get\_type\_extent\_c, 700, 1046 MPI\_FILE\_GET\_VIEW, 658, 659 MPI\_FILE\_IREAD, 659, 675, 688, 710 MPI\_FILE\_IREAD\_ALL, 659, 676, 677, 1051 MPI\_File\_iread\_all\_c, 676, 1046 MPI\_FILE\_IREAD\_AT, 659, 666 MPI\_FILE\_IREAD\_AT\_ALL, 659, 667, 1051 MPI\_File\_iread\_at\_all\_c, 667, 1046 MPI\_File\_iread\_at\_c, 666, 1046 MPI\_File\_iread\_c, <u>675</u>, 1046 MPI\_FILE\_IREAD\_SHARED, 659, 683 MPI\_File\_iread\_shared\_c, <u>683</u>, 1046 MPI\_FILE\_IWRITE, 659, 677 MPI\_FILE\_IWRITE\_ALL, 659, 678, 1051 MPI\_File\_iwrite\_all\_c, 678, 1046 MPI\_FILE\_IWRITE\_AT, 659, 668 MPI\_FILE\_IWRITE\_AT\_ALL, 659, 669, 1051 MPI\_File\_iwrite\_at\_all\_c, 669, 1046 MPI\_File\_iwrite\_at\_c, 668, 1046 MPI\_File\_iwrite\_c, 677, 1046 MPI\_FILE\_IWRITE\_SHARED, 659, 684 MPI\_File\_iwrite\_shared\_c, 684, 1046 MPI\_FILE\_OPEN, 471, 520, 645, 646, 647, 653, 655, 657, 680, 714, 715, 720, 721 MPI\_FILE\_PREALLOCATE, 649, 650, 710, 715 MPI\_FILE\_READ, 659, 660, 670, 671, 672, 675, 714, 715 MPI\_FILE\_READ\_ALL, 659, 672, 677, 689

MPI_FILE_READ_ALL_BEGIN, 16, 659, 688,	1
$689,  \underline{692},  709,  833$	2
MPI_File_read_all_begin_c, 692, 1046	3
MPI_File_read_all_c, <u>672</u> , <u>1046</u>	4
MPI_FILE_READ_ALL_END, 659, 688, 689,	
<u>693</u> , 710, 833	5
MPI_FILE_READ_AT, 659, <u>662</u> , 663, 664, 666	6
	7
MPI_FILE_READ_AT_ALL, 659, <u>663</u> , 664, 667	8
$MFI_FILE_READ_AI_ALL_DEGIN, 10, 009,$	
<u></u> , 000	9
	1(
,,	11
MPI_FILE_READ_AT_ALL_END, 659, 690, 833	12
MPI_File_read_at_c, <u>662</u> , 1046	13
MPI_File_read_c, 670, 1046	
MPI_FILE_READ_ORDERED, 659, 685, 686	14
MPI_FILE_READ_ORDERED_BEGIN, 16, 659,	15
	16
	17
	18
	19
	2(
MPI_FILE_READ_SHARED, 659, <u>681</u> , 683, 686	21
MPI_File_read_shared_c, <u>681</u> , 1046	22
$MPI_FILE_SEEK,  \underline{078},  079$	
$WFI_FILE_SEEK_SHARED, \underline{001}, \underline{000}, \underline{112}$	23
MPI_FILE_SET_ATOMICITY, 647, 710, 711	24
MPI_FILE_SET_ERRHANDLER, 460, 466, 720	25
MPI_FILE_SET_INFO, <u>653</u> , 654, 655, 1047,	26
1062	27
MPL FILE SET SIZE 649 650 710 713 715	
MPI_FILE_SET_VIEW, 136, 472, 646, 653–655,	28
$\underbrace{656}_{657}, 658, 679, 688, 698, 705, 714,$	29
	3(
	31
	32
110	
, , , , , , , ,	33
	34
MPI_FILE_WRITE_ALL_BEGIN, 16, 659, 693,	35
820, 833	36
MPI_File_write_all_begin_c, $\underline{693}$ , $1046$	
MPI_File_write_all_c, <u>674</u> , 1046	37
MPI_FILE_WRITE_ALL_END, 659, 694, 833	38
MPI_FILE_WRITE_AT, 659, 660, 664, 665, 666,	39
	4(
MOLEUE MOLE AT ALL CEO CCE CCC CCO	41
MDI EILE M/DITE AT ALL DECINE 16 650	
691, 833	42
MPI_File_write_at_all_begin_c, <u>691</u> , 1046	43
	44
MPI_File_write_at_all_c, $\underline{665}$ , $1046$	45
$MPI_FILE_WRITE_AI_ALL_END,  \underline{059},  \underline{092}, $	46
000	
	47
MPI_File_write_c, <u>673</u> , 1046	48

1	MPI_FILE_WRITE_ORDERED, 659, 685, <u>686</u>
2	MPI_FILE_WRITE_ORDERED_BEGIN, 16, 659,
3	$\underline{696}, 833$
4	MPI_File_write_ordered_begin_c, 696, 1046
5	MPI_File_write_ordered_c, $\underline{686}$ , $1046$
6	MPI_FILE_WRITE_ORDERED_END, 659, 697,
7	833
8	MPI_FILE_WRITE_SHARED, 659, 660, <u>682</u> ,
9	684-686
	MPI_File_write_shared_c, <u>682</u> , 1046 MPI_FINALIZE, 20, 27, 28, 453, 459, 487, <u>494</u> ,
10	$\begin{array}{c} \text{MP1}_{\text{FINALIZE}}, 20, 27, 28, 453, 459, 487, \underline{494}, \\ 495-499, 502, 518, 546, 547, 646, 731, \end{array}$
11	742, 755, 757, 841, 844, 845, 1048, 742, 755, 757, 841, 844, 845, 1048, 742, 755, 757, 841, 844, 845, 1048, 742, 755, 757, 841, 844, 845, 1048, 742, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 845, 1048, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 844, 845, 1048, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 841, 845, 755, 757, 845, 755, 757, 845, 755, 757, 845, 755, 755, 757, 845, 755, 755, 755, 755, 755, 755, 755, 7
12	1055, 1062
13	MPI_FINALIZED, 497, <u>498</u> , 499, 513, 841, 1051
14	MPI_FREE_MEM, <u>457</u> , 471, 557, 559
15	MPI_GATHER, 187, 190–192, <u>196</u> , 199, 200,
16	207, 208, 214, 225, 255
17	MPI_Gather_c, <u>196</u> , 1046
18	MPI_GATHER_INIT, 187, 191, 192, <u>279</u> , 1046
19	MPI_Gather_init_c, <u>279</u> , 1046
20	MPI_GATHERV, 187, 191, 192, <u>198</u> , 199–201,
21	209, 216, 257
22	MPI_Gatherv_c, <u>198</u> , 1046 MPI_GATHERV_INIT, 187, 191, 192, <u>281</u> , 1046
23	MPI_Gatherv_init_c, <u>281</u> , 1046
24	MPI_GET, 551, 569, <u>573</u> , 574, 582, 587, 593,
25	615, 617, 618, 626, 831, 1063
26	MPI_GET_ACCUMULATE, 551, 569, 578, 580,
27	581, 582, 591, 619, 624, 625, 1050
28	MPI_Get_accumulate_c, <u>580</u> , 1046
29	MPI_GET_ADDRESS, 25, 121, <u>141</u> , 142, 143,
30	157, 563, 787, 804, 822, 827, 846-848
	MPI_Get_c, <u>573</u> , 1046
31	MPI_GET_COUNT, 39, <u>40</u> , 41, 71, 155, 156,
32	584, 641, 662, 1053, 1060
33	MPI_Get_count_c, <u>40</u> , 1046 MPI_GET_ELEMENTS, 71, <u>154</u> , 155, 156, 641,
34	642, 662, 1053
35	MPI_Get_elements_c, <u>154</u> , 1046
36	MPI_GET_ELEMENTS_X, 71, 154, <u>155</u> , 156,
37	641, 662, 1053
38	MPI_GET_LIBRARY_VERSION, <u>452</u> , 513,
39	1049,1051,1053
40	$MPI_GET_PROCESSOR_NAME, \underline{454}, 455, 1062$
41	MPI_GET_VERSION, <u>451</u> , 452, 513, 807, 1051
42	MPI_GRAPH_CREATE, 391, <u>394</u> , 396, 401, 404,
43	408, 416, 417, 1061
44	MPI_GRAPH_GET, <u>391</u> , <u>404</u> MPI_GRAPH_MAP, <u>391</u> , <u>416</u>
45	MPI_GRAPH_MAP, 391, <u>410</u> MPI_GRAPH_NEIGHBORS, 391, <u>408</u> , 409, 417,
46	1059
47	MPI_GRAPH_NEIGHBORS_COUNT, 391, <u>407</u> ,
48	408, 409, 1059

MPI\_GRAPHDIMS\_GET, 391, 404 MPI\_GREQUEST\_COMPLETE, 634-636, 637 MPI\_GREQUEST\_START, <u>634</u>, 875, 878, 1059 MPI\_GROUP\_C2F, <u>842</u> MPI\_GROUP\_COMPARE, 317, 321 MPI\_GROUP\_DIFFERENCE, <u>320</u> MPI\_GROUP\_EXCL, 321, 323 MPI\_GROUP\_F2C, 842 MPI\_GROUP\_FREE, 324, 325, 326, 460, 495, 502, 1062 MPI\_GROUP\_FROM\_SESSION\_PSET, 324, 1048 MPI\_GROUP\_INCL, 320, 321, 322 MPI\_GROUP\_INTERSECTION, 319 MPI\_GROUP\_RANGE\_EXCL, 323 MPI\_GROUP\_RANGE\_INCL, 322 MPI\_GROUP\_RANK, 316, 326 MPI\_GROUP\_SIZE, 316, 326 MPI\_GROUP\_TRANSLATE\_RANKS, 317, 1060 MPI\_GROUP\_UNION, 319 MPI\_IALLGATHER, 187, 191, 192, 260 MPI\_lallgather\_c, 260, 1046 MPI\_IALLGATHERV, 187, 191, 192, <u>262</u> MPI\_lallgatherv\_c, 262, 1046 MPI\_IALLREDUCE, 187, 191, 192, 270 MPI\_lallreduce\_c, <u>270</u>, 1046 MPI\_IALLTOALL, 187, 191, 192, 263 MPI\_lalltoall\_c, <u>263</u>, <u>1046</u> MPI\_IALLTOALLV, 187, 191, 192, 265 MPI\_lalltoallv\_c, <u>265</u>, 1046 MPI\_IALLTOALLW, 187, 191, 192, 267 MPI\_lalltoallw\_c, <u>267</u>, 1046 MPI\_IBARRIER, 187, 191, 192, 250, 252, 305 MPI\_IBCAST, 16, 187, 191, 192, 253, 309 MPI\_lbcast\_c, <u>253</u>, 1046 MPI\_IBSEND, 64, 71, 100 MPI\_lbsend\_c, <u>64</u>, 1046 MPI\_IEXSCAN, 188, 191, 275 MPI\_lexscan\_c, <u>275</u>, 1046 MPI\_IGATHER, 187, 191, 192, 254 MPI\_lgather\_c, 254, 1046 MPI\_IGATHERV, 187, 191, 192, 256 MPI\_lgatherv\_c, 256, 1046 MPI\_IMPROBE, 16, 84, 87, 88, 89, 92, 519, 1049. 1053. 1054 MPI\_IMRECV, 87-90, 91, 92, 1054 MPI\_Imrecv\_c, 91, 1046 MPI\_INEIGHBOR\_ALLGATHER, 392, 429, 1054 MPI\_Ineighbor\_allgather\_c, <u>429</u>, 1046 MPI\_INEIGHBOR\_ALLGATHERV, 392, 431, 1054 MPI\_Ineighbor\_allgatherv\_c, 431, 1046 MPI\_INEIGHBOR\_ALLTOALL, 392, <u>432</u>, 1054 MPI\_Ineighbor\_alltoall\_c, <u>432</u>, 1046

MPI\_INEIGHBOR\_ALLTOALLV, 392, 434, 1054 MPI\_Ineighbor\_alltoallv\_c, 434, 1046 MPI\_INEIGHBOR\_ALLTOALLW, 392, 436, 1054, 1055 MPI\_Ineighbor\_alltoallw\_c, 436, 1046 MPI\_INFO\_C2F, 513, 842 MPI\_INFO\_CREATE, 480, 513 MPI\_INFO\_CREATE\_ENV, <u>484</u>, 513, 1048 MPI\_INFO\_DELETE, 471, 481, 483, 513 MPI\_INFO\_DUP, 483, 513 MPI\_INFO\_F2C, 513, 842 MPI\_INFO\_FREE, 347, 484, 495, 502, 507, 513, 569, 654, 760, 764, 766, 768 MPI\_INFO\_GET, 25, 784, 785, 1048, 1062 MPI\_INFO\_GET\_NKEYS, 479, 482, 483, 513, 1062 MPI\_INFO\_GET\_NTHKEY, 479, 483, 513, 1062 MPI\_INFO\_GET\_STRING, 25, 479, 481, 482, 513, 784, 785, 1047 MPI\_INFO\_GET\_VALUELEN, 25, 785, 786, 1048, 1062 MPI\_INFO\_SET, 480, 481, 483, 513, 785 MPI\_INIT, 20, 27, 28, 315, 453, 459, 484, 487, 488, 489, 491–494, 497–499, 513, 519, 524, 525, 527, 544, 545, 731, 734, 742, 755, 841, 844, 845, 1048, 1053, 1055, 1057.1059 MPI\_INIT\_THREAD, 27, 315, 459, 484, 487-489, 491, 492-494, 497-499, 501, 519, 544, 731, 734, 742, 841, 1053, 1055, 1059 MPI\_INITIALIZED, 497, 498, 513, 841, 1051 MPI\_INTERCOMM\_CREATE, 325, 328, 334, 335, 358, 359, 361, 362, 1054 MPI\_INTERCOMM\_CREATE\_FROM\_GROUPS, 325, 327, 328, 358, <u>360</u>, 453, 1048 MPI\_INTERCOMM\_MERGE, 327, 334, 355, 358, 359, 361, 362, 1056 MPI\_IPROBE, 16, 40, 84, 85-89, 92, 519, 1053 MPI\_IRECV, 16, 67, 92, 821, 822, 825, 826 MPI\_Irecv\_c, 67, 1046 MPI\_IREDUCE, 187, 191, 192, 269, 270 MPI\_Ireduce\_c, 269, 1046 MPI\_IREDUCE\_SCATTER, 188, 191, 192, 273 MPI\_IREDUCE\_SCATTER\_BLOCK, 188, 191, 192, 271 MPI\_Ireduce\_scatter\_block\_c, 272, 1046 MPI\_Ireduce\_scatter\_c, 273, 1046 MPI\_IRSEND, 66 MPI\_Irsend\_c, 66, 1046 MPI\_IS\_THREAD\_MAIN, 491, 494, 1051 MPI\_ISCAN, 188, 191, 274 MPI\_lscan\_c, 274, 1046 MPI\_ISCATTER, 187, 191, 192, 257

MPI_lscatter_c, <u>258</u> , 1046	1
MPI_ISCATTERV, 187, 191, 192, <u>259</u>	2
MPI_lscatterv_c, <u>259</u> , 1046	3
MPI_ISEND, 16, <u>63</u> , 100, 799, 800, 803, 820,	4
821, 826, 827	5
MPI_Isend_c, <u>63</u> , 1046	
MPI_ISENDRECV, <u>68</u> , <u>1046</u>	6
MPI_Isendrecv_c, <u>68</u> , <u>1046</u>	7
MPI_ISENDRECV_REPLACE, <u>69</u> , 1046	8
MPI_Isendrecv_replace_c, <u>69</u> , <u>1046</u>	9
MPI_ISSEND, 65	10
MPI_Issend_c, <u>65</u> , 1046	11
MPI_KEYVAL_CREATE, 25, <u>781</u> , 783, 879	12
MPI_KEYVAL_FREE, 25, 379, 782	
MPI_LOOKUP_NAME, 471, 534, 539, 540, 541	13
MPI_MESSAGE_C2F, <u>842</u> , 1054	14
MPI_MESSAGE_F2C, 842, 1054	15
MPI_MPROBE, 16, 84, 87, <u>89</u> , 92, 519, 1053,	16
1054	17
MPI_MRECV, 16, 87–89, <u>90</u> , 91, 92, 1054	18
MPI_Mrecv_c, <u>90</u> , 1046	19
MPI_NEIGHBOR_ALLGATHER, 392, 418, 420,	20
422, 430, 1045, 1054	
MPI_Neighbor_allgather_c, <u>418</u> , <u>1046</u>	21
MPI_NEIGHBOR_ALLGATHER_INIT, 438, 1046	22
MPI_Neighbor_allgather_init_c, <u>438</u> , 1046	23
MPI_NEIGHBOR_ALLGATHERV, 392, 420,	24
432,1045,1054	25
MPI_Neighbor_allgatherv_c, <u>421</u> , 1046	26
MPI_NEIGHBOR_ALLGATHERV_INIT, <u>439</u> ,	27
799,1046,1049	28
MPI_Neighbor_allgatherv_init_c, <u>439</u> , 1046	
MPI_NEIGHBOR_ALLTOALL, 392, <u>422</u> , 424,	29
425,  433,  1045,  1054	30
MPI_Neighbor_alltoall_c, <u>422</u> , 1046	31
MPI_NEIGHBOR_ALLTOALL_INIT, <u>441</u> , 1046	32
MPI_Neighbor_alltoall_init_c, <u>441</u> , <u>1046</u>	33
MPI_NEIGHBOR_ALLTOALLV, 392, <u>425</u> , 435,	34
1045,1054	35
MPI_Neighbor_alltoallv_c, $\underline{425}$ , $1046$	36
MPI_NEIGHBOR_ALLTOALLV_INIT, <u>442</u> , 799,	
1046,1049	37
MPI_Neighbor_alltoallv_init_c, <u>442</u> , <u>1046</u>	38
MPI_NEIGHBOR_ALLTOALLW, <u>392</u> , <u>427</u> , 437,	39
1045,1054,1055	40
MPI_Neighbor_alltoallw_c, $\underline{427}$ , $1046$	41
$MPI\_NEIGHBOR\_ALLTOALLW\_INIT,  \underline{444}, 799,$	42
1046,1049	43
MPI_Neighbor_alltoallw_init_c, <u>444</u> , <u>1046</u>	44
MPI_NULL_COPY_FN, 25, 368, <u>782</u> , 866	
MPI_NULL_DELETE_FN, 25, 369, <u>782</u> , 866	45
MPI_OP_C2F, <u>842</u>	46
MPI_OP_COMMUTATIVE, <u>242</u> , 1058	47
	48

#### 1096

1	MPI_OP_CREATE, <u>233</u> , 234, 236, 799, 873,
2	876, 1056
3	MPI_Op_create_c, 233, 234, 840, 873, 1046
	MPI_OP_F2C, <u>842</u>
4	
5	MPI_OP_FREE, <u>237</u> , 495, 502
6	MPI_OPEN_PORT, 534, <u>535</u> , 537–539, 541
7	MPI_PACK, 60, <u>175</u> , 177, 179, 182, 701, 707
	MPI_Pack_c, <u>175</u> , 1046
8	MPI_PACK_EXTERNAL, 8, <u>182</u> , 814, 1060
9	MPI_Pack_external_c, <u>182</u> , 1046
10	MPI_PACK_EXTERNAL_SIZE, <u>184</u>
11	MPI_Pack_external_size_c, <u>184</u> , <u>1046</u>
12	MPI_PACK_SIZE, 60, <u>178</u> , 179, 1053
	MPI_Pack_size_c, <u>178</u> , 1046
13	MPI_PARRIVED, 104, 110, 111, 1046
14	MPI_PCONTROL, 726, 727, 728
15	MPI_PREADY, 104, 106, <u>108</u> , 109–112, 1046
16	MPI_PREADY_LIST, <u>110</u> , 1046
17	MPI_PREADY_RANGE, <u>109</u> , 110, 1046
	MPI_PRECV_INIT, 104, <u>107</u> , 108, 112, 1046
18	
19	MPI_PROBE, 16, 38, 40, 41, 84, <u>85</u> , 87, 89, 91,
20	92, 519, 1053
21	MPI_PSEND_INIT, 104, <u>106</u> , 107–109, 112,
22	1046
23	MPI_PUBLISH_NAME, 534, <u>538</u> , 539–541
	MPI_PUT, 551, 569, <u>570</u> , 574, 578, 586, 593,
24	599,609,611,616,617,626,820,831,
25	1063
26	MPI_Put_c, <u>571</u> , 1046
27	MPI_QUERY_THREAD, <u>493</u> , 494, 519, 1051
28	MPI_RACCUMULATE, 551, 569, 578, 582, <u>588</u> ,
	589
29	MPI_Raccumulate_c, <u>588</u> , <u>1046</u>
30	MPI_RECV, 16, 32, <u>37</u> , 39–41, 84, 87, 88, 91,
31	$120, 153, 1\overline{54}, 177, 188, 197, 306, 642,$
32	716, 755, 827, 830, 831
33	MPI_Recv_c, <u>37</u> , 1046
34	MPI_RECV_INIT, 16, <u>99</u> , 108
	MPI_Recv_init_c, <u>99</u> , <u>1046</u>
35	MPI_REDUCE, 187, 191, 192, <u>224</u> , 225, 226,
36	234–236, 239, 243, 245, 247, 248, 270,
37	578, 581-583, 1059
38	MPI_Reduce_c, <u>224</u> , 1046
39	
	MPI_REDUCE_INIT, 187, 191, <u>294</u> , 1046
40	MPI_Reduce_init_c, <u>294</u> , 1046
41	MPI_REDUCE_LOCAL, 225, 226, 234, <u>241</u> ,
42	1056, 1058
43	MPI_Reduce_local_c, <u>241</u> , 1046
44	MPI_REDUCE_SCATTER, 188, 191, 192, 226,
	$234, \underline{244}, 245, 274$
45	MPI_REDUCE_SCATTER_BLOCK, 188, 191,
46	$192, 226, 234, \underline{242}, 243, 244, 272, 1058$
47	MPI_Reduce_scatter_block_c, <u>242</u> , 1046
48	

MPI\_REDUCE\_SCATTER\_BLOCK\_INIT, 188, 191, 192, 296, 1046 MPI\_Reduce\_scatter\_block\_init\_c, 296, 1046 MPI\_Reduce\_scatter\_c, 244, 1046 MPI\_REDUCE\_SCATTER\_INIT, 188, 191, 192, 298, 1046 MPI\_Reduce\_scatter\_init\_c, 298, 1046 MPI\_REGISTER\_DATAREP, 471, 704, 705-708, 721, 876, 878 MPI\_Register\_datarep\_c, 704, 708, 840, 876, 1046 MPI\_REQUEST\_C2F, 842 MPI\_REQUEST\_F2C, 842 MPI\_REQUEST\_FREE, 73, 74, 92, 101, 251, 276, 277, 495, 502, 584, 636, 637, 1058 MPI\_REQUEST\_GET\_STATUS, 41, 83, 635, 1058 MPI\_RGET, 551, 569, 586, 587 MPI\_RGET\_ACCUMULATE, 551, 569, 578, 581, <u>590</u>, 591 MPI\_Rget\_accumulate\_c, 590, 1046 MPI\_Rget\_c, 586, 1046 MPI\_RPUT, 551, 569, 585, 586 MPI\_Rput\_c, 585, 1046 MPI\_RSEND, 16, 52 MPI\_Rsend\_c, 52, 1046 MPI\_RSEND\_INIT, 98 MPI\_Rsend\_init\_c, <u>98</u>, <u>1046</u> MPI\_SCAN, 188, 191, 226, 234, 246, 248, 275 MPI\_Scan\_c, 246, 1046 MPI\_SCAN\_INIT, 188, 191, 299, 1046 MPI\_Scan\_init\_c, <u>299</u>, 1046 MPI\_SCATTER, 187, 191, 192, 206, 207, 209, 210, 243, 258 MPI\_Scatter\_c, 207, 1046 MPI\_SCATTER\_INIT, 187, 191, 192, 283, 1046 MPI\_Scatter\_init\_c, <u>283</u>, 1046 MPI\_SCATTERV, 187, 191, 192, 208, 209, 210, 245, 260 MPI\_Scatterv\_c, 209, 1046 MPI\_SCATTERV\_INIT, 187, 191, 192, 284, 1046 MPI\_Scatterv\_init\_c, 285, 1046 MPI\_SEND, 16, 31, 32, 33, 41, 46, 120, 153, 175, 306, 646, 716, 728, 827, 828, 830, 831 MPI\_Send\_c, 32, 1046 MPI\_SEND\_INIT, 16, 95, 100, 107 MPI\_Send\_init\_c, <u>95</u>, 1046 MPI\_SENDRECV, 42, 101, 412 MPI\_Sendrecv\_c, 43, 1046 MPI\_SENDRECV\_REPLACE, 44, 101 MPI\_Sendrecv\_replace\_c, 44, 1046

MPI\_SESSION\_C2F, 842, 1048

MPI\_SESSION\_CALL\_ERRHANDLER, 476, 477, 513, 1048 MPI\_SESSION\_CREATE\_ERRHANDLER, 460, 466, 468, 501, 513, 875, 878, 1048 MPI\_SESSION\_F2C, 842, 1048 MPI\_SESSION\_FINALIZE, 502, 503, 513, 1048 MPI\_SESSION\_GET\_ERRHANDLER, 460, 468, 1048 MPI\_SESSION\_GET\_INFO, 501, 506, 1048 MPI\_SESSION\_GET\_NTH\_PSET, 324, 505, 506.1048 MPI\_SESSION\_GET\_NUM\_PSETS, 504, 505, 1048 MPI\_SESSION\_GET\_PSET\_INFO, 507, 1048 MPI\_SESSION\_INIT, 460, 495, 501, 513, 519, 544, 735, 1048 MPI\_SESSION\_SET\_ERRHANDLER, 460, 467, 1048 MPI\_SIZEOF, 25, 786, 1049 MPI\_SSEND, 51 MPI\_Ssend\_c, 51, 1046 MPI\_SSEND\_INIT, 97 MPI\_Ssend\_init\_c, 97, 1046 MPI\_START, 99, 100, 101, 103, 106, 109, 110, 276, 277, 827 MPI\_STARTALL, <u>100</u>, 101, 103, 106, 109, 110, 276, 277, 827 MPI\_STATUS\_C2F, 844 MPI\_STATUS\_C2F08, 844, 1055 MPI\_STATUS\_F082C, 844, 1055 MPI\_STATUS\_F082F, 846, 1045, 1055 MPI\_STATUS\_F2C, 843 MPI\_STATUS\_F2F08, 845, 1045, 1055 MPI\_STATUS\_SET\_CANCELLED, 641, 1048 MPI\_STATUS\_SET\_ELEMENTS, 640, 641 MPI\_STATUS\_SET\_ELEMENTS\_X, 641, 1053 MPI\_T\_CATEGORY\_CHANGED, 778, 1048 MPI\_T\_CATEGORY\_GET\_CATEGORIES, 777, 778.779 MPI\_T\_CATEGORY\_GET\_CVARS, 776, 778, 779 MPI\_T\_CATEGORY\_GET\_EVENTS, 777, 778, 1048MPI\_T\_CATEGORY\_GET\_INDEX, 776, 779, 1051MPI\_T\_CATEGORY\_GET\_INFO, 775, 778, 779, 1050, 1051 MPI\_T\_CATEGORY\_GET\_NUM, 774 MPI\_T\_CATEGORY\_GET\_NUM\_EVENTS, 775, 776, 1048 MPI\_T\_CATEGORY\_GET\_PVARS, 777, 778, 779 MPI\_T\_CVAR\_GET\_INDEX, <u>740</u>, 741, 779, 1051

MPI_T_CVAR_GET_INFO, 736, 738, <u>739</u> , 740,	1
742-744, 778, 779, 1050, 1051	2
MPI_T_CVAR_GET_NUM, <u>738</u> , 743	3
MPI_T_CVAR_HANDLE_ALLOC, 736, <u>742</u> ,	4
743, 744, 779 MPI_T_CVAR_HANDLE_FREE, 743, 779	5
MPI_T_CVAR_HANDLE_FREE, <u>145</u> , 119 MPI_T_CVAR_READ, <u>743</u> , 779	6
MPI_T_CVAR_WRITE, <u>743</u> , 779	7
MPI_T_ENUM_GET_INFO, 736, <u>737</u> , 779	8
MPI_T_ENUM_GET_ITEM, <u>737</u> , 779	9
MPI_T_EVENT_CALLBACK_GET_INFO, 768,	10
1048	11
MPI_T_EVENT_CALLBACK_SET_INFO, <u>768</u> ,	
1048	12
MPI_T_EVENT_COPY, 761, 764, 772, 1048	13
MPI_T_EVENT_GET_INDEX, <u>764</u> , 765, 779,	14
1048	15
MPI_T_EVENT_GET_INFO, 736, 762, <u>763</u> ,	16
765, 772, 778, 779, 1048	17
MPI_T_EVENT_GET_NUM, <u>762</u> , 1048	18
MPI_T_EVENT_GET_SOURCE, 761, 773, 1048	19
MPI_T_EVENT_GET_TIMESTAMP, 759, 761,	20
$\underline{772},1048$	20
MPI_T_EVENT_HANDLE_ALLOC, <u>765</u> , 768,	
779,1048	22
MPI_T_EVENT_HANDLE_FREE, <u>769</u> , 779,	23
1048	24
MPI_T_EVENT_HANDLE_GET_INFO, <u>766</u> ,	25
1048	26
MPI_T_EVENT_HANDLE_SET_INFO, <u>766</u> ,	27
1048	28
MPI_T_EVENT_READ, 761, 763, 771, 1048	29
MPI_T_EVENT_REGISTER_CALLBACK, <u>767</u> ,	30
	31
MPI_T_EVENT_SET_DROPPED_HANDLER,	
<u>770</u> , 771, 1048	32
MPI_T_FINALIZE, 735	33
MPI_T_INIT_THREAD, <u>734</u> , 735 MPI_T_PVAR_GET_INDEX, <u>749</u> , 779, 1051	34
MPI_T_PVAR_GET_INDEX, <u>149</u> , 179, 1051 MPI_T_PVAR_GET_INFO, 736, 747, <u>748</u> , 751,	35
753, 755, 778, 779, 1050, 1051	36
MPI_T_PVAR_GET_NUM, 747, 751	37
$MPI_T_PVAR_GET_NOM, \underline{141}, 731$ $MPI_T_PVAR_HANDLE\_ALLOC, 736, 751, \mathbf{75$	38
753, 779	39
MPI_T_PVAR_HANDLE_FREE, <u>751</u> , 752, 779,	40
1050	
MPI_T_PVAR_READ, <u>753</u> , 754, 761, 779, 1050	41
MPI_T_PVAR_READRESET, 749, <u>754</u> , 761,	42
779, 1050	43
MPI_T_PVAR_RESET, <u>754</u> , 761, 779, 1050	44
MPI_T_PVAR_SESSION_CREATE, <u>750</u> , 779	45
MPI_T_PVAR_SESSION_FREE, <u>750</u> , 779	46
MPI_T_PVAR_START, <u>752</u> , 761, 779, 1050	47
MPI_T_PVAR_STOP, <u>752</u> , 761, 779, 1050	48

1	MPI_T_PVAR_WRITE, <u>753</u> , 754, 761, 779, 1050
2	MPI_T_SOURCE_GET_INFO, <u>759</u> , 773, 779,
3	1048
4	MPI_T_SOURCE_GET_NUM, <u>758</u> , 779, 1048
	MPI_T_SOURCE_GET_TIMESTAMP, <u>760</u> , 761,
5	779, 1048
6	MPI_TEST, 41, 70, 71, <u>72</u> , 73–75, 77, 83, 92,
7	93, 101, 103, 104, 110–112, 277, 495,
8	637, 660, 661
9	MPI_TEST_CANCELLED, 71, 72, <u>93</u> , 94, 635,
10	642, 662
11	MPI_TESTALL, 75, <u>79</u> , 519, 635–637, 640
	MPI_TESTANY, 75, <u>77</u> , 81, 519, 635, 636, 640
12	MPI_TESTSOME, 75, <u>81</u> , 82, 519, 635–637, 640
13	MPI_TOPO_TEST, 391, <u>403</u>
14	MPI_TYPE_C2F, <u>842</u>
15	MPI_TYPE_COMMIT, <u>151</u> , 843
16	MPI_TYPE_CONTIGUOUS, 17, <u>121</u> , 124, 145,
17	
	159, 644, 700
18	MPI_Type_contiguous_c, <u>121</u> , 1046
19	MPI_TYPE_CREATE_DARRAY, 17, 41, <u>136</u> ,
20	137, 159
21	MPI_Type_create_darray_c, <u>136</u> , 1046
22	MPI_TYPE_CREATE_F90_COMPLEX, 17, 159,
23	162, 227, 704, 792, <u>811</u> , 813, 814
	MPI_TYPE_CREATE_F90_INTEGER, 17, 159,
24	162, 227, 704, 792, <u>812</u> , 813, 814
25	MPI_TYPE_CREATE_F90_REAL, 17, 159, 162,
26	227, 704, 792, <u>811</u> , 812–814, 1058
27	MPI_TYPE_CREATE_HINDEXED, 17, 25, 121,
28	127, <u>128</u> , 131, 133, 159, 787, 839
29	MPI_TYPE_CREATE_HINDEXED_BLOCK, 17,
30	$121, \underline{130}, 159, 839, 1054$
	MPI_Type_create_hindexed_block_c, <u>130</u> , 1046
31	MPI_Type_create_hindexed_c, <u>128</u> , 1046
32	MPI_TYPE_CREATE_HVECTOR, 17, 25, 121,
33	$\underline{124}, 159, 787$
34	MPI_Type_create_hvector_c, <u>124</u> , 1046
35	MPI_TYPE_CREATE_INDEXED_BLOCK, 17,
36	$\underline{129}, 130, 159$
37	MPI_Type_create_indexed_block_c, <u>129</u> , 1046
	MPI_TYPE_CREATE_KEYVAL, <u>365</u> , <u>376</u> , <u>379</u> ,
38	849,874,877,1060
39	MPI_TYPE_CREATE_RESIZED, 25, 121, 145,
40	$\underline{148}, 149, 159, 700, 788, 1056$
41	MPI_Type_create_resized_c, <u>148</u> , 1046
42	MPI_TYPE_CREATE_STRUCT, 17, 25, 121,
43	$\underline{131}, 132, 133, 145, 159, 222, 787, 839$
	MPI_Type_create_struct_c, <u>131</u> , 1046
44	MPI_TYPE_CREATE_SUBARRAY, 17, 20, <u>133</u> ,
45	135, 138, 159
46	MPI_Type_create_subarray_c, <u>133</u> , 1046
47	MPI_TYPE_DELETE_ATTR, 365, <u>379</u> , 1056
48	MPI_TYPE_DUP, 17, <u>152</u> , 159, 1056

MPI_TYPE_DUP_FN, <u>376</u> , 865, 1051
MPI_TYPE_EXTENT, 25, <u>787</u> , 1052
MPI_TYPE_F2C, <u>842</u>
MPI_TYPE_FREE, <u>151</u> , 161, 377, 495, 502
MPI_TYPE_FREE_KEYVAL, 365, <u>377</u> , 379
MPI_TYPE_GET_ATTR, <u>365</u> , <u>378</u> , <u>379</u> , <u>800</u> ,
849, 1056
MPI_TYPE_GET_CONTENTS, 158, 159, <u>160</u> ,
161–163, 839
MPI_Type_get_contents_c, <u>160</u> , <u>1046</u>
MPI_TYPE_GET_ENVELOPE, <u>158</u> , 159, 161,
162,813,839
MPI_Type_get_envelope_c, <u>158</u> , <u>1046</u>
MPI_TYPE_GET_EXTENT, 25, <u>147</u> , 150, 787,
815, 846
MPI_Type_get_extent_c, <u>147</u> , 1046
MPI_TYPE_GET_EXTENT_X, <u>147</u> , 1053
MPI_TYPE_GET_NAME, <u>385</u> , <u>1056</u>
MPI_TYPE_GET_TRUE_EXTENT, <u>149</u>
MPI_Type_get_true_extent_c, <u>149</u> , <u>1046</u>
MPI_TYPE_GET_TRUE_EXTENT_X, 149, 150
1053
MPI_TYPE_HINDEXED, 25, <u>787</u> , 1052
MPI_TYPE_HVECTOR, 25, <u>787</u> , 1052
MPI_TYPE_INDEXED, 17, <u>126</u> , 127, 129, 159
$MPI\_Type\_indexed\_c, \underline{126}, \underline{1046}$
MPI_TYPE_LB, 25, <u>787</u> , 1052
MPI_TYPE_MATCH_SIZE, 792, 815, 1056
MPI_TYPE_NULL_COPY_FN, <u>376</u> , 865, 1051
MPI_TYPE_NULL_DELETE_FN, <u>376</u> , 865,
1051, 1056
MPI_TYPE_SET_ATTR, 365, <u>378</u> , 379, 800,
849, 853, 1056
MPI_TYPE_SET_NAME, <u>384</u> , 1056
MPI_TYPE_SIZE, <u>143</u> , 144, 728, 1053
MPI_Type_size_c, <u>143</u> , 1046 MPI_TYPE_SIZE_X, <u>144</u> , 1053
MPI_1YPE_SIZE_X, <u>144</u> , 1053
MPI_TYPE_STRUCT, 25, <u>787</u> , 1052
MPI_TYPE_UB, 25, <u>787</u> , 1052
MPI_TYPE_VECTOR, 17, 122, <u>123</u> , 124, 127,
159
MPI_Type_vector_c, <u>123</u> , 1046
MPI_UNPACK, <u>176</u> , <u>177</u> , 182, 707
MPI_Unpack_c, <u>176</u> , 1046
MPI_UNPACK_EXTERNAL, 8, <u>183</u> , 814
MPI_Unpack_external_c, <u>183</u> , 1046
MPI_UNPUBLISH_NAME, 472, <u>540</u>
MPI_WAIT, 39, 41, 70, <u>71</u> , 72–76, 78, 92, 93,
101, 103, 104, 110-112, 250, 277, 306,
495, 519, 633, 637, 660, 661, 688, 710,
711, 820, 826, 827, 830
MPI_WAITALL, 75, <u>78</u> , 79, 251, 307, 519, 584,
635-637, 640

MPI\_WAITANY, 54, 75, <u>76</u>, 82, 519, 635, 636, 640 MPI\_WAITSOME, 75, 80, 81, 82, 519, 635-637, 640 MPI\_WIN\_ALLOCATE, 552, 556, 557, 559, 565, 567, 572, 605, 805, 806, 1047, 1049 MPI\_Win\_allocate\_c, 556, 1046 MPI\_WIN\_ALLOCATE\_CPTR, 557, 1049 MPI\_WIN\_ALLOCATE\_SHARED, 339, 552, 558, 559, 561, 565, 567, 605, 806, 1047 - 1049MPI\_Win\_allocate\_shared\_c, 558, 1046 MPI\_WIN\_ALLOCATE\_SHARED\_CPTR, 559, 1049 MPI\_WIN\_ATTACH, 562, 563, 564, 565, 605 MPI\_WIN\_C2F, 842 MPI\_WIN\_CALL\_ERRHANDLER, 475, 477 MPI\_WIN\_COMPLETE, 565, 594, 598, 599-602, 610, 617 MPI\_WIN\_CREATE, 520, 552, 553, 555, 557, 559, 563-565, 567, 609 MPI\_Win\_create\_c, 553, 1046 MPI\_WIN\_CREATE\_DYNAMIC, 472, 552, 562, 563, 564, 566, 567, 610 MPI\_WIN\_CREATE\_ERRHANDLER, 460, 463, 464, 875, 877, 1056 MPI\_WIN\_CREATE\_KEYVAL, 365, 372, 379, 849, 874, 877, 1060 MPI\_WIN\_DELETE\_ATTR, 365, 375, 379 MPI\_WIN\_DETACH, 562, 564, 566 MPI\_WIN\_DUP\_FN, <u>373</u>, 865, 1051 MPI\_WIN\_F2C, 842 MPI\_WIN\_FENCE, 565, 574, 594, 596, 597, 598, 607, 608, 610, 611, 614, 619, 831 MPI\_WIN\_FLUSH, 560, 584, 586, 605, 606, 610, 625, 626 MPI\_WIN\_FLUSH\_ALL, 584, 586, 606, 610 MPI\_WIN\_FLUSH\_LOCAL, 584, 606, 610 MPI\_WIN\_FLUSH\_LOCAL\_ALL, 584, 607, 610 MPI\_WIN\_FREE, 374, 495, 502, 547, 565, 566 MPI\_WIN\_FREE\_KEYVAL, 365, 374, 379 MPI\_WIN\_GET\_ATTR, 365, 375, 379, 566, 800, 849, 852 MPI\_WIN\_GET\_ERRHANDLER, 460, 464, 1062 MPI\_WIN\_GET\_GROUP, 567 MPI\_WIN\_GET\_INFO, 567, 568, 569, 1047, 1054MPI\_WIN\_GET\_NAME, 386 MPI\_WIN\_LOCK, 554, 565, 594, 602, 603-605, 607, 609, 611, 614-616 MPI\_WIN\_LOCK\_ALL, 554, 594, 603, 604, 607, 609, 611, 616, 625 MPI\_WIN\_NULL\_COPY\_FN, <u>373</u>, 865, 1051 MPI\_WIN\_NULL\_DELETE\_FN, <u>373</u>, 865, 1051

MPI_WIN_POST, 565, 594, 598, <u>599</u> , 600–602,	1
604,607,608,611,617,619	2
MPI_WIN_SET_ATTR, 365, <u>374</u> , 379, 566, 800,	3
849,853	4
MPI_WIN_SET_ERRHANDLER, 460, 464	5
MPI_WIN_SET_INFO, <u>568</u> , 1047, 1054	6
MPI_WIN_SET_NAME, <u>385</u>	
MPI_WIN_SHARED_QUERY, 559, <u>560</u> , 806,	7
1049	8
MPI_Win_shared_query_c, <u>560</u> , 1046	9
MPI_WIN_SHARED_QUERY_CPTR, 562, 1049	10
MPI_WIN_START, 565, 594, 598, 599-602, 607,	11
608,617,624	12
MPI_WIN_SYNC, <u>607</u> , 611–614, 617, 624, 626	
MPI_WIN_TEST, <u>600</u> , 601	13
MPI_WIN_UNLOCK, 565, 586, 594, <u>603</u> , 605,	14
610,611,614,615	15
MPI_WIN_UNLOCK_ALL, 586, 594, 603, 604,	16
610, 611, 614, 625	17
MPI_WIN_WAIT, 565, 594, <u>600</u> , 601, 602, 604,	18
610, 611, 614, 617, 618	19
MPI_WTICK, 26, <u>478</u> , 760	
MPI_WTIME, 16, 26, 454, <u>477</u> , 478, 728, 747,	20
760	21
mpiexec, 489, 490, 492, <u>515</u> , 1048	22
mpirun, $515$	23
inpiruti, oro	$^{24}$
PMPI_, 725, 800	25
PMPI_AINT_ADD, 26	
PMPI_AINT_DIFF, 26	26
PMPI_ISEND, 800, 803	27
	28
PMPI_WTICK, 26	28 29
PMPI_WTICK, 26	29
PMPI_WTICK, 26	29 30 31
PMPI_WTICK, 26	29 30 31 32
PMPI_WTICK, 26	29 30 31
PMPI_WTICK, 26	29 30 31 32
PMPI_WTICK, 26	29 30 31 32 33
PMPI_WTICK, 26	29 30 31 32 33 34
PMPI_WTICK, 26	29 30 31 32 33 34 35
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40 41
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40 41 42
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
PMPI_WTICK, 26	29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46