This document describes the Message-Passing Interface (MPI) standard, version 2.1. The MPI standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for C, C++, and Fortran are defined.

Technically, this version of the standard is based on “MPI: A Message-Passing Interface Standard, June 12, 1995” (MPI-1.1) from the MPI-1 Forum, and “MPI-2: Extensions to the Message-Passing Interface, July, 1997” (MPI-1.2 and MPI-2.0) from the MPI-2 Forum, and errata documents from the MPI Forum.

Historically, the evolution of the standards is from MPI-1.0 (June 1994) to MPI-1.1 (June 12, 1995) to MPI-1.2 (July 18, 1997), with several clarifications and additions and published as part of the MPI-2 document, to MPI-2.0 (July 18, 1997), with new functionality, to MPI-1.3 (May 30, 2008), combining for historical reasons the documents 1.1 and 1.2 and some errata documents to one combined document, and this document, MPI-2.1, combining the previous documents. Additional clarifications and errata corrections to MPI-2.0 are also included.
The goal of the Message-Passing Interface, simply stated, is to develop a widely used practical, portable, efficient, and flexible standard for message-passing programs. As such the interface should establish a standard for writing message-passing programs. This copy of the document contains all the technical features proposed for the interface. The changes from Version 1.0 are minor. A version of this document with all changes marked is available. This paragraph is an example of a change.

Additional errata and clarifications collected by the MPI Forum are also included in this document. Additional errata collected by the MPI Forum referring to MPI-1.1 and MPI-1.2 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 don’t fit elsewhere, in particular language interoperability.

Version 2.1: June 23, 2008, 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 don’t 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 contains clarifications and minor corrections to Version 1.1 of the MPI Standard. The only 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.

Version 1.1: June, 1995. Beginning in March, 1995, the Message-Passing Interface Forum reconvened to correct errors and make clarifications in the MPI document of May 5, 1994, referred to below as Version 1.0. These discussions resulted in Version 1.1, which is this document. The changes from Version 1.0 are minor. A version of this document with all changes marked is available. This paragraph is an example of a change.

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

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.

This is the final report, Version 1.0, of the Message-Passing Interface Forum. This document contains all the technical features proposed for the interface. This copy of the draft was processed by LATEX on May 5, 1994.

Please send comments on MPI to mpi-comments@mpi-forum.org. Your comment will be forwarded to MPI Forum committee members who will attempt to respond.
# Contents

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>xvii</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction to MPI</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview and Goals</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Background of MPI-1.0</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Background of MPI-1.3 and MPI-2.1</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Who Should Use This Standard?</td>
<td>4</td>
</tr>
<tr>
<td>1.6 What Platforms Are Targets For Implementation?</td>
<td>4</td>
</tr>
<tr>
<td>1.7 What Is Included In The Standard?</td>
<td>5</td>
</tr>
<tr>
<td>1.8 What Is Not Included In The Standard?</td>
<td>5</td>
</tr>
<tr>
<td>1.9 Organization of this Document</td>
<td>6</td>
</tr>
<tr>
<td>2 MPI Terms and Conventions</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Document Notation</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Naming Conventions</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Procedure Specification</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Semantic Terms</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Data Types</td>
<td>12</td>
</tr>
<tr>
<td>2.5.1 Opaque Objects</td>
<td>12</td>
</tr>
<tr>
<td>2.5.2 Array Arguments</td>
<td>14</td>
</tr>
<tr>
<td>2.5.3 State</td>
<td>14</td>
</tr>
<tr>
<td>2.5.4 Named Constants</td>
<td>14</td>
</tr>
<tr>
<td>2.5.5 Choice</td>
<td>15</td>
</tr>
<tr>
<td>2.5.6 Addresses</td>
<td>15</td>
</tr>
<tr>
<td>2.5.7 File Offsets</td>
<td>15</td>
</tr>
<tr>
<td>2.6 Language Binding</td>
<td>15</td>
</tr>
<tr>
<td>2.6.1 Deprecated Names and Functions</td>
<td>16</td>
</tr>
<tr>
<td>2.6.2 Fortran Binding Issues</td>
<td>16</td>
</tr>
<tr>
<td>2.6.3 C Binding Issues</td>
<td>18</td>
</tr>
<tr>
<td>2.6.4 C++ Binding Issues</td>
<td>18</td>
</tr>
<tr>
<td>2.6.5 Functions and Macros</td>
<td>21</td>
</tr>
<tr>
<td>2.7 Processes</td>
<td>22</td>
</tr>
<tr>
<td>2.8 Error Handling</td>
<td>22</td>
</tr>
<tr>
<td>2.9 Implementation Issues</td>
<td>23</td>
</tr>
<tr>
<td>2.9.1 Independence of Basic Runtime Routines</td>
<td>23</td>
</tr>
<tr>
<td>2.9.2 Interaction with Signals</td>
<td>24</td>
</tr>
</tbody>
</table>

# Contents

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>xvii</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction to MPI</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview and Goals</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Background of MPI-1.0</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Background of MPI-1.3 and MPI-2.1</td>
<td>3</td>
</tr>
<tr>
<td>1.5 Who Should Use This Standard?</td>
<td>4</td>
</tr>
<tr>
<td>1.6 What Platforms Are Targets For Implementation?</td>
<td>4</td>
</tr>
<tr>
<td>1.7 What Is Included In The Standard?</td>
<td>5</td>
</tr>
<tr>
<td>1.8 What Is Not Included In The Standard?</td>
<td>5</td>
</tr>
<tr>
<td>1.9 Organization of this Document</td>
<td>6</td>
</tr>
<tr>
<td>2 MPI Terms and Conventions</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Document Notation</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Naming Conventions</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Procedure Specification</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Semantic Terms</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Data Types</td>
<td>12</td>
</tr>
<tr>
<td>2.5.1 Opaque Objects</td>
<td>12</td>
</tr>
<tr>
<td>2.5.2 Array Arguments</td>
<td>14</td>
</tr>
<tr>
<td>2.5.3 State</td>
<td>14</td>
</tr>
<tr>
<td>2.5.4 Named Constants</td>
<td>14</td>
</tr>
<tr>
<td>2.5.5 Choice</td>
<td>15</td>
</tr>
<tr>
<td>2.5.6 Addresses</td>
<td>15</td>
</tr>
<tr>
<td>2.5.7 File Offsets</td>
<td>15</td>
</tr>
<tr>
<td>2.6 Language Binding</td>
<td>15</td>
</tr>
<tr>
<td>2.6.1 Deprecated Names and Functions</td>
<td>16</td>
</tr>
<tr>
<td>2.6.2 Fortran Binding Issues</td>
<td>16</td>
</tr>
<tr>
<td>2.6.3 C Binding Issues</td>
<td>18</td>
</tr>
<tr>
<td>2.6.4 C++ Binding Issues</td>
<td>18</td>
</tr>
<tr>
<td>2.6.5 Functions and Macros</td>
<td>21</td>
</tr>
<tr>
<td>2.7 Processes</td>
<td>22</td>
</tr>
<tr>
<td>2.8 Error Handling</td>
<td>22</td>
</tr>
<tr>
<td>2.9 Implementation Issues</td>
<td>23</td>
</tr>
<tr>
<td>2.9.1 Independence of Basic Runtime Routines</td>
<td>23</td>
</tr>
<tr>
<td>2.9.2 Interaction with Signals</td>
<td>24</td>
</tr>
</tbody>
</table>
3 Point-to-Point Communication 25

3.1 Introduction 25
3.2 Blocking Send and Receive Operations 26
  3.2.1 Blocking Send 26
  3.2.2 Message Data 27
  3.2.3 Message Envelope 28
  3.2.4 Blocking Receive 29
  3.2.5 Return Status 31
  3.2.6 Passing MPI_STATUS_IGNORE for Status 33
3.3 Data Type Matching and Data Conversion 34
  3.3.1 Type Matching Rules 34
  3.3.2 Data Conversion 36
3.4 Communication Modes 38
3.5 Semantics of Point-to-Point Communication 41
3.6 Buffer Allocation and Usage 45
  3.6.1 Model Implementation of Buffered Mode 47
3.7 Nonblocking Communication 47
  3.7.1 Communication Request Objects 49
  3.7.2 Communication Initiation 49
  3.7.3 Communication Completion 52
  3.7.4 Semantics of Nonblocking Communications 56
  3.7.5 Multiple Completions 57
  3.7.6 Non-destructive Test of status 63
3.8 Probe and Cancel 64
3.9 Persistent Communication Requests 68
3.10 Send-Receive 73
3.11 Null Processes 75

4 Datatypes 77

4.1 Derived Datatypes 77
  4.1.1 Type Constructors with Explicit Addresses 79
  4.1.2 Datatype Constructors 79
  4.1.3 Subarray Datatype Constructor 87
  4.1.4 Distributed Array Datatype Constructor 89
  4.1.5 Address and Size Functions 94
  4.1.6 Lower-Bound and Upper-Bound Markers 96
  4.1.7 Extent and Bounds of Datatypes 98
  4.1.8 True Extent of Datatypes 98
  4.1.9 Commit and Free 99
  4.1.10 Duplicating a Datatype 100
  4.1.11 Use of General Datatypes in Communication 101
  4.1.12 Correct Use of Addresses 103
  4.1.13 Decoding a Datatype 104
  4.1.14 Examples 111
4.2 Pack and Unpack 120
5 Collective Communication 129
5.1 Introduction and Overview .................................................. 129
5.2 Communicator Argument ...................................................... 132
  5.2.1 Specifics for Intracomputer Collective Operations ................. 132
  5.2.2 Applying Collective Operations to Intercommunicators .......... 133
  5.2.3 Specifics for Intercommunicator Collective Operations .......... 134
5.3 Barrier Synchronization .................................................... 135
5.4 Broadcast ........................................................................ 136
  5.4.1 Example using MPI_BCAST .......................................... 136
5.5 Gather ............................................................................ 137
  5.5.1 Examples using MPI_GATHER, MPI_GATHERV .................. 140
5.6 Scatter ............................................................................ 147
  5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV ............... 149
5.7 Gather-to-all .................................................................... 152
  5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV ........ 154
5.8 All-to-All Scatter/Gather ..................................................... 155
5.9 Global Reduction Operations ................................................. 159
  5.9.1 Reduce ..................................................................... 160
  5.9.2 Predefined Reduction Operations ................................. 161
  5.9.3 Signed Characters and Reductions ............................... 163
  5.9.4 MINLOC and MAXLOC ............................................ 164
  5.9.5 User-Defined Reduction Operations ............................ 168
  Example of User-defined Reduce ..................................... 170
  5.9.6 All-Reduce .............................................................. 171
5.10 Reduce-Scatter ................................................................. 173
5.11 Scan .............................................................................. 174
  5.11.1 Inclusive Scan ........................................................ 174
  5.11.2 Exclusive Scan ........................................................ 175
  5.11.3 Example using MPI_SCAN ......................................... 176
5.12 Correctness .................................................................. 177

6 Groups, Contexts, Communicators, and Caching 181
6.1 Introduction ...................................................................... 181
  6.1.1 Features Needed to Support Libraries ............................. 181
6.2 Basic Concepts .................................................................. 182
  6.2.1 Groups .................................................................... 184
  6.2.2 Contexts ................................................................... 184
  6.2.3 Intra-Communicators ............................................... 185
  6.2.4 Predefined Intra-Communicators .................................. 185
6.3 Group Management ............................................................. 186
  6.3.1 Group Accessors ....................................................... 186
  6.3.2 Group Constructors ................................................... 187
  6.3.3 Group Destorctors ..................................................... 188
6.4 Communicator Management ................................................. 193
  6.4.1 Communicator Accessors ............................................. 193

4.3 Canonical MPI_PACK and MPI_UNPACK .................................. 127

5 Collective Communication 129
5.1 Introduction and Overview .................................................. 129
5.2 Communicator Argument ...................................................... 132
  5.2.1 Specifics for Intracomputer Collective Operations ................. 132
  5.2.2 Applying Collective Operations to Intercommunicators .......... 133
  5.2.3 Specifics for Intercommunicator Collective Operations .......... 134
5.3 Barrier Synchronization .................................................... 135
5.4 Broadcast ........................................................................ 136
  5.4.1 Example using MPI_BCAST .......................................... 136
5.5 Gather ............................................................................ 137
  5.5.1 Examples using MPI_GATHER, MPI_GATHERV .................. 140
5.6 Scatter ............................................................................ 147
  5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV ............... 149
5.7 Gather-to-all .................................................................... 152
  5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV ........ 154
5.8 All-to-All Scatter/Gather ..................................................... 155
5.9 Global Reduction Operations ................................................. 159
  5.9.1 Reduce ..................................................................... 160
  5.9.2 Predefined Reduction Operations ................................. 161
  5.9.3 Signed Characters and Reductions ............................... 163
  5.9.4 MINLOC and MAXLOC ............................................ 164
  5.9.5 User-Defined Reduction Operations ............................ 168
  Example of User-defined Reduce ..................................... 170
  5.9.6 All-Reduce .............................................................. 171
5.10 Reduce-Scatter ................................................................. 173
5.11 Scan .............................................................................. 174
  5.11.1 Inclusive Scan ........................................................ 174
  5.11.2 Exclusive Scan ........................................................ 175
  5.11.3 Example using MPI_SCAN ......................................... 176
5.12 Correctness .................................................................. 177

6 Groups, Contexts, Communicators, and Caching 181
6.1 Introduction ...................................................................... 181
  6.1.1 Features Needed to Support Libraries ............................. 181
  6.1.2 MPI’s Support for Libraries ......................................... 182
6.2 Basic Concepts .................................................................. 184
  6.2.1 Groups .................................................................... 184
  6.2.2 Contexts ................................................................... 184
  6.2.3 Intra-Communicators ............................................... 185
  6.2.4 Predefined Intra-Communicators .................................. 185
6.3 Group Management ............................................................. 186
  6.3.1 Group Accessors ....................................................... 186
  6.3.2 Group Constructors ................................................... 187
  6.3.3 Group Destorctors ..................................................... 188
6.4 Communicator Management ................................................. 193
  6.4.1 Communicator Accessors ............................................. 193
8 MPI Environmental Management 259
8.1 Implementation Information 259
8.1.1 Version Inquiries 259
8.1.2 Environmental Inquiries 260
8.1.2.1 Tag Values 260
8.1.2.2 Host Rank 260
8.1.2.3 IO Rank 261
8.1.2.4 Clock Synchronization 261
8.2 Memory Allocation 262
8.3 Error Handling 264
8.3.1 Error Handlers for Communicators 265
8.3.2 Error Handlers for Windows 267
8.3.3 Error Handlers for Files 269
8.3.4 Freeing Errorhandlers and Retrieving Error Strings 270
8.4 Error Codes and Classes 271
8.5 Error Classes, Error Codes, and Error Handlers 273
8.6 Timers and Synchronization 277
8.7 Startup 278
8.7.1 Allowing User Functions at Process Termination 283
8.7.2 Determining Whether MPI Has Finished 284
8.8 Portable MPI Process Startup 284
9 The Info Object 287
10 Process Creation and Management 293
10.1 Introduction 293
10.2 The Dynamic Process Model 294
10.2.1 Starting Processes 294
10.2.2 The Runtime Environment 294
10.3 Process Manager Interface 296
10.3.1 Processes in MPI 296
10.3.2 Starting Processes and Establishing Communication 296
10.3.3 Starting Multiple Executables and Establishing Communication 301
10.3.4 Reserved Keys 303
10.3.5 Spawn Example 304
10.3.5.1 Simple Example — Completely Portable. 313
10.3.5.2 Ocean/Atmosphere - Relies on Name Publishing 314
10.3.5.3 Simple Client-Server Example. 314
10.4 Establishing Communication 306
10.4.1 Names, Addresses, Ports, and All That 306
10.4.2 Server Routines 308
10.4.3 Client Routines 309
10.4.4 Name Publishing 311
10.4.5 Reserved Key Values 313
10.4.6 Client/Server Examples 313
10.5 Other Functionality 316
10.5.1 Universe Size 316
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.5.2 Singleton MPI_INIT</td>
<td>316</td>
</tr>
<tr>
<td>10.5.3 MPI_APPNUM</td>
<td>317</td>
</tr>
<tr>
<td>10.5.4 Releasing Connections</td>
<td>318</td>
</tr>
<tr>
<td>10.5.5 Another Way to Establish MPI Communication</td>
<td>319</td>
</tr>
<tr>
<td>11 One-Sided Communications</td>
<td>321</td>
</tr>
<tr>
<td>11.1 Introduction</td>
<td>321</td>
</tr>
<tr>
<td>11.2 Initialization</td>
<td>322</td>
</tr>
<tr>
<td>11.2.1 Window Creation</td>
<td>322</td>
</tr>
<tr>
<td>11.2.2 Window Attributes</td>
<td>324</td>
</tr>
<tr>
<td>11.3 Communication Calls</td>
<td>325</td>
</tr>
<tr>
<td>11.3.1 Put</td>
<td>326</td>
</tr>
<tr>
<td>11.3.2 Get</td>
<td>328</td>
</tr>
<tr>
<td>11.3.3 Examples</td>
<td>328</td>
</tr>
<tr>
<td>11.3.4 Accumulate Functions</td>
<td>331</td>
</tr>
<tr>
<td>11.4 Synchronization Calls</td>
<td>333</td>
</tr>
<tr>
<td>11.4.1 Fence</td>
<td>333</td>
</tr>
<tr>
<td>11.4.2 General Active Target Synchronization</td>
<td>339</td>
</tr>
<tr>
<td>11.4.3 Lock</td>
<td>342</td>
</tr>
<tr>
<td>11.4.4 Assertions</td>
<td>344</td>
</tr>
<tr>
<td>11.4.5 Miscellaneous Clarifications</td>
<td>346</td>
</tr>
<tr>
<td>11.5 Examples</td>
<td>346</td>
</tr>
<tr>
<td>11.6 Error Handling</td>
<td>348</td>
</tr>
<tr>
<td>11.6.1 Error Handlers</td>
<td>348</td>
</tr>
<tr>
<td>11.6.2 Error Classes</td>
<td>349</td>
</tr>
<tr>
<td>11.7 Semantics and Correctness</td>
<td>349</td>
</tr>
<tr>
<td>11.7.1 Atomicity</td>
<td>352</td>
</tr>
<tr>
<td>11.7.2 Progress</td>
<td>352</td>
</tr>
<tr>
<td>11.7.3 Registers and Compiler Optimizations</td>
<td>354</td>
</tr>
<tr>
<td>12 External Interfaces</td>
<td>357</td>
</tr>
<tr>
<td>12.1 Introduction</td>
<td>357</td>
</tr>
<tr>
<td>12.2 Generalized Requests</td>
<td>357</td>
</tr>
<tr>
<td>12.2.1 Examples</td>
<td>361</td>
</tr>
<tr>
<td>12.3 Associating Information with Status</td>
<td>363</td>
</tr>
<tr>
<td>12.4 MPI and Threads</td>
<td>365</td>
</tr>
<tr>
<td>12.4.1 General</td>
<td>365</td>
</tr>
<tr>
<td>12.4.2 Clarifications</td>
<td>366</td>
</tr>
<tr>
<td>12.4.3 Initialization</td>
<td>368</td>
</tr>
<tr>
<td>13 I/O</td>
<td>373</td>
</tr>
<tr>
<td>13.1 Introduction</td>
<td>373</td>
</tr>
<tr>
<td>13.1.1 Definitions</td>
<td>373</td>
</tr>
<tr>
<td>13.2 File Manipulation</td>
<td>375</td>
</tr>
<tr>
<td>13.2.1 Opening a File</td>
<td>375</td>
</tr>
<tr>
<td>13.2.2 Closing a File</td>
<td>377</td>
</tr>
<tr>
<td>13.2.3 Deleting a File</td>
<td>378</td>
</tr>
<tr>
<td>13.2.4 Resizing a File</td>
<td>379</td>
</tr>
<tr>
<td>10.5.2 Singleton MPI_INIT</td>
<td>316</td>
</tr>
<tr>
<td>10.5.3 MPI_APPNUM</td>
<td>317</td>
</tr>
<tr>
<td>10.5.4 Releasing Connections</td>
<td>318</td>
</tr>
<tr>
<td>10.5.5 Another Way to Establish MPI Communication</td>
<td>319</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Page</th>
<th>Figure Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Collective communications, an overview</td>
</tr>
<tr>
<td>5.2</td>
<td>Intercommunicator allgather</td>
</tr>
<tr>
<td>5.3</td>
<td>Intercommunicator reduce-scatter</td>
</tr>
<tr>
<td>5.4</td>
<td>Gather example</td>
</tr>
<tr>
<td>5.5</td>
<td>Gatherv example with strides</td>
</tr>
<tr>
<td>5.6</td>
<td>Gatherv example, 2-dimensional</td>
</tr>
<tr>
<td>5.7</td>
<td>Gatherv example, 2-dimensional, subarrays with different sizes</td>
</tr>
<tr>
<td>5.8</td>
<td>Gatherv example, 2-dimensional, subarrays with different sizes and strides</td>
</tr>
<tr>
<td>5.9</td>
<td>Scatterv example</td>
</tr>
<tr>
<td>5.10</td>
<td>Scatterv example with strides</td>
</tr>
<tr>
<td>5.11</td>
<td>Scatterv example with different strides and counts</td>
</tr>
<tr>
<td>5.12</td>
<td>Race conditions with point-to-point and collective communications</td>
</tr>
<tr>
<td>6.1</td>
<td>Intercommunicator create using MPI_COMM_CREATE</td>
</tr>
<tr>
<td>6.2</td>
<td>Intercommunicator construction with MPI_COMM_SPLIT</td>
</tr>
<tr>
<td>6.3</td>
<td>Three-group pipeline</td>
</tr>
<tr>
<td>6.4</td>
<td>Three-group ring</td>
</tr>
<tr>
<td>7.1</td>
<td>Set-up of process structure for two-dimensional parallel Poisson solver</td>
</tr>
<tr>
<td>11.1</td>
<td>Active target communication</td>
</tr>
<tr>
<td>11.2</td>
<td>Active target communication, with weak synchronization</td>
</tr>
<tr>
<td>11.3</td>
<td>Passive target communication</td>
</tr>
<tr>
<td>11.4</td>
<td>Active target communication with several processes</td>
</tr>
<tr>
<td>11.5</td>
<td>Schematic description of window</td>
</tr>
<tr>
<td>11.6</td>
<td>Symmetric communication</td>
</tr>
<tr>
<td>11.7</td>
<td>Deadlock situation</td>
</tr>
<tr>
<td>11.8</td>
<td>No deadlock</td>
</tr>
<tr>
<td>13.1</td>
<td>Etypes and filetypes</td>
</tr>
<tr>
<td>13.2</td>
<td>Partitioning a file among parallel processes</td>
</tr>
<tr>
<td>13.3</td>
<td>Displacements</td>
</tr>
<tr>
<td>13.4</td>
<td>Example array file layout</td>
</tr>
<tr>
<td>13.5</td>
<td>Example local array filetype for process 1</td>
</tr>
</tbody>
</table>

# List of Figures

<table>
<thead>
<tr>
<th>Page</th>
<th>Figure Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Collective communications, an overview</td>
</tr>
<tr>
<td>5.2</td>
<td>Intercommunicator allgather</td>
</tr>
<tr>
<td>5.3</td>
<td>Intercommunicator reduce-scatter</td>
</tr>
<tr>
<td>5.4</td>
<td>Gather example</td>
</tr>
<tr>
<td>5.5</td>
<td>Gatherv example with strides</td>
</tr>
<tr>
<td>5.6</td>
<td>Gatherv example, 2-dimensional</td>
</tr>
<tr>
<td>5.7</td>
<td>Gatherv example, 2-dimensional, subarrays with different sizes</td>
</tr>
<tr>
<td>5.8</td>
<td>Gatherv example, 2-dimensional, subarrays with different sizes and strides</td>
</tr>
<tr>
<td>5.9</td>
<td>Scatterv example</td>
</tr>
<tr>
<td>5.10</td>
<td>Scatterv example with strides</td>
</tr>
<tr>
<td>5.11</td>
<td>Scatterv example with different strides and counts</td>
</tr>
<tr>
<td>5.12</td>
<td>Race conditions with point-to-point and collective communications</td>
</tr>
<tr>
<td>6.1</td>
<td>Intercommunicator create using MPI_COMM_CREATE</td>
</tr>
<tr>
<td>6.2</td>
<td>Intercommunicator construction with MPI_COMM_SPLIT</td>
</tr>
<tr>
<td>6.3</td>
<td>Three-group pipeline</td>
</tr>
<tr>
<td>6.4</td>
<td>Three-group ring</td>
</tr>
<tr>
<td>7.1</td>
<td>Set-up of process structure for two-dimensional parallel Poisson solver</td>
</tr>
<tr>
<td>11.1</td>
<td>Active target communication</td>
</tr>
<tr>
<td>11.2</td>
<td>Active target communication, with weak synchronization</td>
</tr>
<tr>
<td>11.3</td>
<td>Passive target communication</td>
</tr>
<tr>
<td>11.4</td>
<td>Active target communication with several processes</td>
</tr>
<tr>
<td>11.5</td>
<td>Schematic description of window</td>
</tr>
<tr>
<td>11.6</td>
<td>Symmetric communication</td>
</tr>
<tr>
<td>11.7</td>
<td>Deadlock situation</td>
</tr>
<tr>
<td>11.8</td>
<td>No deadlock</td>
</tr>
<tr>
<td>13.1</td>
<td>Etypes and filetypes</td>
</tr>
<tr>
<td>13.2</td>
<td>Partitioning a file among parallel processes</td>
</tr>
<tr>
<td>13.3</td>
<td>Displacements</td>
</tr>
<tr>
<td>13.4</td>
<td>Example array file layout</td>
</tr>
<tr>
<td>13.5</td>
<td>Example local array filetype for process 1</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Deprecated constructs ............................................. 17
3.1 Predefined MPI datatypes corresponding to Fortran datatypes ........ 27
3.2 Predefined MPI datatypes corresponding to C datatypes ............ 28
4.1 combiner values returned from MPI_TYPE_GET_ENVELOPE ............ 106
6.1 MPI_COMM_* Function Behavior (in Inter-Communication Mode) ....... 211
8.1 Error classes (Part 1) ........................................... 272
8.2 Error classes (Part 2) ........................................... 273
11.1 Error classes in one-sided communication routines .................. 349
13.1 Data access routines ........................................... 388
13.2 "external32" sizes of predefined datatypes .......................... 416
13.3 I/O Error Classes ............................................ 431
16.1 C++ names for the MPI C and C++ predefined datatypes ............. 454
16.2 C++ names for the MPI Fortran predefined datatypes ............... 454
16.3 C++ names for other MPI datatypes ................................ 455
2.1 Deprecated constructs ............................................. 17
3.1 Predefined MPI datatypes corresponding to Fortran datatypes ........ 27
3.2 Predefined MPI datatypes corresponding to C datatypes ............ 28
4.1 combiner values returned from MPI_TYPE_GET_ENVELOPE ............ 106
6.1 MPI_COMM_* Function Behavior (in Inter-Communication Mode) ....... 211
8.1 Error classes (Part 1) ........................................... 272
8.2 Error classes (Part 2) ........................................... 273
11.1 Error classes in one-sided communication routines .................. 349
13.1 Data access routines ........................................... 388
13.2 "external32" sizes of predefined datatypes .......................... 416
13.3 I/O Error Classes ............................................ 431
16.1 C++ names for the MPI C and C++ predefined datatypes ............. 454
16.2 C++ names for the MPI Fortran predefined datatypes ............... 454
16.3 C++ names for other MPI datatypes ................................ 455
Chapter 9

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, MPI::Info in C++, and INTEGER in Fortran. 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.

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.

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 it is an argument to a non-blocking routine, info is parsed before that routine returns, so that it may be modified or freed immediately after return.

When the descriptions refer to a key or value as being a boolean, an integer, or a list, they mean the string representation of these types. An implementation may define its own rules for how info value strings are converted to other types, but to ensure portability, every implementation must support the following representations. Legal values for a boolean must include the strings “true” and “false” (all lowercase). For integers, legal values must include the strings “true” and “false” (all lowercase). F or integers, legal values must include the strings “true” and “false” (all lowercase).
string representations of decimal values of integers that are within the range of a standard integer type in the program. (However it is possible that not every legal integer is a legal value for a given key.) On positive numbers, + signs are optional. No space may appear between a + or – sign and the leading digit of a number. For comma separated lists, the string must contain legal elements separated by commas. Leading and trailing spaces are stripped automatically from the types of info values described above and for each element of a comma separated list. These rules apply to all info values of these types. Implementations are free to specify a different interpretation for values of other info keys.

MPI_INFO_CREATE(info)
OUT info info object created (handle)

int MPI_Info_create(MPI_Info *info)
MPI_INFO_CREATE(INFO, IERROR)
INTEGER INFO, IERROR

static MPI::Info MPI::Info::Create()

MPI_INFO_CREATE creates a new info object. The newly created object contains no key/value pairs.

MPI_INFO_SET(info, key, value)
INOUT info info object (handle)
IN key key (string)
IN value value (string)

int MPI_Info_set(MPI_Info info, char *key, char *value)
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
INTEGER INFO, IERRORCHARACTER*(*) KEY, VALUE

does the (key, value) pair to info, and overrides the value if a value for the same key was previously set. key and value are null-terminated strings in C. In Fortran, leading and trailing spaces in key and value are stripped. If either key or value are larger than the allowed maximums, the errors MPI_ERR_INFO_KEY or MPI_ERR_INFO_VALUE are raised, respectively.

MPI_INFO_DELETE(info, key)
INOUT info info object (handle)
IN key key (string)

int MPI_Info_delete(MPI_Info info, char *key)

MPI_INFO_DELETE deletes the key from the info object. If the key is not present, the operation has no effect.
MPI_INFO_DELETE(INFO, KEY, IERROR)
INTEGER INFO, IERROR
CHARACTER*(*) KEY
void MPI::Info::Delete(const char* key)
MPI_INFO_DELETE deletes a (key,value) pair from info. If key is not defined in info, the call raises an error of class MPI_ERR_INFO_NOKEY.

MPI_INFO_GET(info, key, valuelen, value, flag)
IN info info object (handle)
IN key key (string)
IN valuelen length of value arg (integer)
OUT value value (string)
OUT flag true if key defined, false if not (boolean)

int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value, int *flag)
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
INTEGER INFO, VALUELEN, IERROR
CHARACTER*(*) KEY, VALUE
LOGICAL FLAG

bool MPI::Info::Get(const char* key, int valuelen, char* value) const
This function retrieves the value associated with key in a previous call to MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. value is the number of characters available in value. If it is less than the actual size of the value, the value is truncated. In C, valuelen should be one less than the amount of allocated space to allow for the null terminator.

If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.

MPI_INFO_GET_VALUELEN(info, key, valuelen, flag)
IN info info object (handle)
IN key key (string)
OUT valuelen length of value arg (integer)
OUT flag true if key defined, false if not (boolean)

int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen, int *flag)
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
INTEGER INFO, VALUELEN, IERROR
LOGICAL FLAG

This function retrieves the value associated with key in a previous call to MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. value is the number of characters available in value. If it is less than the actual size of the value, the value is truncated. In C, valuelen should be one less than the amount of allocated space to allow for the null terminator.

If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.
bool MPI::Info::Get_valuelen(const char* key, int& valuelen) const

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 or C++ does not include the end-of-string character.

If key is larger than MPI_MAX_INFO_KEY, the call is erroneous.

MPI_INFO_GET_NKEYS(info, nkeys)
IN info info object (handle)
OUT nkeys number of defined keys (integer)

int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)

MPI_INFO_GET_NTHKEY(info, n, key)
IN info info object (handle)
IN n key number (integer)
OUT key key (string)

int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)

MPI_INFO_DUP(info, newinfo)
IN info info object (handle)
OUT newinfo info object (handle)

int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
MPI_INFO_DUP(INFO, NEWINFO, IERROR)
INTEGER INFO, NEWINFO, IERROR
MPI::Info MPI::Info::Dup() const

MPI_INFO_DUP duplicates an existing info object, creating a new object, with the same (key, value) pairs and the same ordering of keys.

MPI_INFO_FREE(info)
INOUT info info object (handle)

int MPI_Info_free(MPI_Info *info)

MPI_INFO_FREE(info)
INOUT info info object (handle)

 MPI_INFO_FREE(MPI_INFO, IERROR)
INTEGER INFO, IERROR
void MPI::Info::Free()

This function frees info and sets it to MPI_INFO_NULL. The value of an info argument is interpreted each time the info is passed to a routine. Changes to an info after return from a routine do not affect that interpretation.
Chapter 10

Process Creation and Management

10.1 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 allow for a variety of approaches to process management while placing minimal restrictions on the execution environment.

The MPI model for process creation allows both the creation of an initial set of processes related by their membership in a common MPI_COMM_WORLD and the creation and management of processes after an MPI application has been started. A major impetus for the later form of process creation comes from the PVM [23] research effort. This work has provided a wealth of experience with process management and resource control that illustrates their benefits and potential pitfalls.

The MPI Forum decided not to address resource control because it was not able to design a portable interface that would be appropriate for the broad spectrum of existing and potential resource and process controllers. Resource control can encompass a wide range of abilities, including adding and deleting nodes from a virtual parallel machine, reserving and scheduling resources, managing compute partitions of an MPP, and returning information about available resources. Assumes that resource control is provided externally — probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

The reasons for including process management in MPI are both technical and practical. Important classes of message-passing applications require process control. These include task farms, serial applications with parallel modules, and problems that require a run-time assessment of the number and type of processes that should be started. On the practical side, users of workstation clusters who are migrating from PVM to MPI may be accustomed to using PVM’s capabilities for process and resource management. The lack of these features would be a practical stumbling block to migration.

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. These include everything from tightly integrated MPPs to heterogeneous networks of workstations.
- 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 process management 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.

10.2 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.

10.2.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager, which can range from a parallel operating system (CMOST) to layered software (POE) to an `rsh` command (`p4`).

`MPI_COMM_SPAWN` starts MPI processes and establishes communication with them, returning an intercommunicator. `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 intercommunicator.

MPI uses the existing group abstraction to represent processes. A process is identified by a (group, rank) pair.

10.2.2 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. Examples of such environments are:

- MPP managed by a batch queueing system. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.

The process management 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.

10.2 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.

10.2.1 Starting Processes

MPI applications may start new processes through an interface to an external process manager, which can range from a parallel operating system (CMOST) to layered software (POE) to an `rsh` command (`p4`).

`MPI_COMM_SPAWN` starts MPI processes and establishes communication with them, returning an intercommunicator. `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 intercommunicator.

MPI uses the existing group abstraction to represent processes. A process is identified by a (group, rank) pair.

10.2.2 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. Examples of such environments are:

- MPP managed by a batch queueing system. Batch queueing systems generally allocate resources before an application begins, enforce limits on resource use (CPU time, memory use, etc.), and do not allow a change in resource allocation after a job begins. Moreover, many MPPs have special limitations or extensions, such as a limit on the number of processes that may run on one processor, or the ability to gang-schedule processes of a parallel application.
Network of workstations with PVM. PVM (Parallel Virtual Machine) allows a user to create a “virtual machine” out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.

Network of workstations managed by a load balancing system. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.

Large SMP with Unix. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

—COMM—MPI

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 environment-specific API. An example of such an API would be the PVM task and machine management routines — pvm_addhosts, pvm_config, pvm_tasks, etc., possibly modified to return an MPI (group,rank) when possible. A Condor or PBS API would be another possibility.

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 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:

- 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.

—MULTIPLE_SPAWNCOMM—MPI

—MULTIPLE_SPAWNCOMM—MPI

Network of workstations with PVM. PVM (Parallel Virtual Machine) allows a user to create a “virtual machine” out of a network of workstations. An application may extend the virtual machine or manage processes (create, kill, redirect output, etc.) through the PVM library. Requests to manage the machine or processes may be intercepted and handled by an external resource manager.

Network of workstations managed by a load balancing system. A load balancing system may choose the location of spawned processes based on dynamic quantities, such as load average. It may transparently migrate processes from one machine to another when a resource becomes unavailable.

Large SMP with Unix. Applications are run directly by the user. They are scheduled at a low level by the operating system. Processes may have special scheduling characteristics (gang-scheduling, processor affinity, deadline scheduling, processor locking, etc.) and be subject to OS resource limits (number of processes, amount of memory, etc.).

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 environment-specific API. An example of such an API would be the PVM task and machine management routines — pvm_addhosts, pvm_config, pvm_tasks, etc., possibly modified to return an MPI (group,rank) when possible. A Condor or PBS API would be another possibility.

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 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:

- 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 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.

### 10.3 Process Manager Interface

#### 10.3.1 Processes in MPI

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.

#### 10.3.2 Starting Processes and Establishing Communication

The following routine starts a number of MPI processes and establishes communication with them, returning an intercommunicator.

**Advice to users.** It is possible in MPI to start a static SPMD or MPMD application by starting first 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.)*

```c
MPI\_COMM\_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array\_of\_errcodes)
```

| IN | command | name of program to be spawned (string, significant only at root) |
| IN | argv | arguments to command (array of strings, significant only at root) |
| IN | maxprocs | maximum number of processes to start (integer, significant only at root) |
| IN | info | a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root) |
| IN | root | rank of process in which previous arguments are examined (integer) |
| IN | comm | intracommunicator containing group of spawning processes (handle) |
| OUT | intercomm | intercommunicator between original group and the newly spawned group (handle) |
| OUT | array\_of\_errcodes | one code per process (array of integer) |

**Advice to users.** It is possible in MPI to start a static SPMD or MPMD application by starting first 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.)*

```c
MPI\_COMM\_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array\_of\_errcodes)
```

| IN | command | name of program to be spawned (string, significant only at root) |
| IN | argv | arguments to command (array of strings, significant only at root) |
| IN | maxprocs | maximum number of processes to start (integer, significant only at root) |
| IN | info | a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root) |
| IN | root | rank of process in which previous arguments are examined (integer) |
| IN | comm | intracommunicator containing group of spawning processes (handle) |
| OUT | intercomm | intercommunicator between original group and the newly spawned group (handle) |
| OUT | array\_of\_errcodes | one code per process (array of integer) |
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[])

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root) const

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[] )

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, char* argv[], int maxprocs, const MPI::Info& info, int root)

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*) , IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[] )

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root)

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*) , IERROR

1.3. PROCESS MANAGER INTERFACE

int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

character(*) COMMAND, ARGV(*)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[] )

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root)

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

character(*) COMMAND, ARGV(*)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*) , IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[] )

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root)

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

character(*) COMMAND, ARGV(*)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*) , IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[] )

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root)

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

character(*) COMMAND, ARGV(*)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*) , IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[] )

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root)

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR)

character(*) COMMAND, ARGV(*)

integer INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*) , IERROR

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root, int array_of_errcodes[] )

MPI::Intercomm MPI::Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const MPI::Info& info, int root)
under POE on an IBM SP would use POE's method of finding executables. 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.)

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.)

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 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, C++ and Fortran to indicate an empty argument list. In C and C++, this constant is the same as NULL.

Example 10.1 Examples of argv in C and Fortran
To run the program “ocean” with arguments “-gridfile” and “ocean1.grd” in C:

```c
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Comm_spawn(command, argv, ...);
```

or, if not everything is known at compile time:

```c
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:

```fortran
CHARACTER*25 command, argv(3)
command = 'ocean'
argv(1) = ' -gridfile '
argv(2) = ' ocean1.grd'
argv(3) = ' '
call MPI_COMM_SPAWN(command, argv, ...)
```

Example 10.1 Examples of argv in C and Fortran
To run the program “ocean” with arguments “-gridfile” and “ocean1.grd” in C:

```c
char command[] = "ocean";
char *argv[] = {"-gridfile", "ocean1.grd", NULL};
MPI_Comm_spawn(command, argv, ...);
```

or, if not everything is known at compile time:

```c
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:

```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[2]` of `main` corresponds to `argv[0]` in `MPI_COMM_SPAWN`. `argv[2]` of `main` to `argv[1]` of `MPI_COMM_SPAWN`, etc. Second, `argv` of `MPI_COMM_SPAWN` must be null-terminated, so that its length can be determined. Passing an `argv` of `MPI_ARGV_NULL` to `MPI_COMM_SPAWN` results in `main` receiving `argv` of 1 and an `argv` whose element 0 is (conventionally) the name of the program.

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 \( \{ m_i : 0 \leq m_i \leq \text{maxprocs} \} \) 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, `MPI_COMM_SPAWN` returns successfully and the number of spawned processes, \( m \), is given by the size of the remote group of intercomm. If \( m \) is less than `maxprocs`, reasons why the other processes were not spawned are given in `array_of_encodes` as described below. If it is not possible to spawn one of the allowed numbers of processes, `MPI_COMM_SPAWN` raises 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 10.3.4 on page 303 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 \( \{ 0 \cdots N \} \). However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)

The info argument. The info argument to all of the routines in this chapter is an opaque handle of type `MPI_Info` in C, `MPI_Info` in C++ and `INTEGER` in Fortran. It is a container for a number of user-specified (key,value) pairs. `key` and `value` are strings (null-terminated `char*` in C, `character(*)` in Fortran). Routines to create and manipulate the info argument are described in Section 9 on page 287.

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`.

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[2]` of `main` corresponds to `argv[0]` in `MPI_COMM_SPAWN`. `argv[2]` of `main` to `argv[1]` of `MPI_COMM_SPAWN`, etc. Second, `argv` of `MPI_COMM_SPAWN` must be null-terminated, so that its length can be determined. Passing an `argv` of `MPI_ARGV_NULL` to `MPI_COMM_SPAWN` results in `main` receiving `argv` of 1 and an `argv` whose element 0 is (conventionally) the name of the program.

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 \( \{ m_i : 0 \leq m_i \leq \text{maxprocs} \} \) 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, `MPI_COMM_SPAWN` returns successfully and the number of spawned processes, \( m \), is given by the size of the remote group of intercomm. If \( m \) is less than `maxprocs`, reasons why the other processes were not spawned are given in `array_of_encodes` as described below. If it is not possible to spawn one of the allowed numbers of processes, `MPI_COMM_SPAWN` raises 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 10.3.4 on page 303 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 \( \{ 0 \cdots N \} \). However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)

The info argument. The info argument to all of the routines in this chapter is an opaque handle of type `MPI_Info` in C, `MPI_Info` in C++ and `INTEGER` in Fortran. It is a container for a number of user-specified (key,value) pairs. `key` and `value` are strings (null-terminated `char*` in C, `character(*)` in Fortran). Routines to create and manipulate the info argument are described in Section 9 on page 287.

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`. 
MPI does not specify the content of the info argument, except to reserve a number of special key values (see Section 10.3.4 on page 303). The info argument is quite flexible and could even be used, for example, to specify the executable and its command-line arguments. In this case the command argument to MPI_COMM_SPAWN could be empty. The ability to do this follows from the fact that MPI does not specify how an executable is found, and the info argument can tell the runtime system where to “find” the executable (empty string). Of course a program that does this will not be portable across MPI implementations.

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.

The array of errcodes argument. The array of errcodes is an array of length maxprocs in which MPI reports the status of each process that MPI was requested to start. If all maxprocs processes were spawned, array of errcodes is filled in with the value MPI_SUCCESS. If only m (0 ≤ m < maxprocs) processes are spawned, m of the entries will contain MPI_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes. In C++ this constant does not exist, and the array of errcodes argument may be omitted from the argument list.

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 on page 14. (End of advice to implementors.)

MPI_COMM_GET_PARENT

OUT parent the parent communicator (handle)

int MPI_Comm_get_parent(MPI_Comm *parent)

MPI_COMM_GET_PARENT(PARENT, IERROR)

INTEGER PARENT, IERROR

static MPI::Intercomm MPI::Comm::Get_parent()

If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, MPI_COMM_GET_PARENT returns the “parent” intercommunicator of the current process. This parent intercommunicator is created implicitly inside of MPI_INIT and is the same intercommunicator 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.
10.3. PROCESS MANAGER INTERFACE

Advice to users. MPI_COMM_GET_PARENT returns a handle to a single intercommunicator. Calling MPI_COMM_GET_PARENT a second time returns a handle to the same intercommunicator. Freeing the handle with MPI_COMM_DISCONNECT or MPI_COMM_FREE will cause other references to the intercommunicator to become invalid (dangling). Note that calling MPI_COMM_FREE on the parent communicator is not useful. (End of advice to users.)

Rationale. 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. (End of rationale.)

10.3.3 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.


IN count number of commands (positive integer, significant to MPI only at root — see advice to users)
IN array_of_commands programs to be executed (array of strings, significant only at root)
IN array_of_argv arguments for commands (array of arrays of strings, significant only at root)
IN array_of_maxprocs maximum number of processes to start for each command (array of integer, significant only at root)
IN array_of_info info objects telling the runtime system where and how to start processes (array of handles, significant only at root)
IN root rank of process in which previous arguments are examined (integer)
IN comm intracommunicator containing group of spawning processes (handle)
OUT intercomm intercommunicator between original group and newly spawned group (handle)
OUT array_of_errcodes one error code per process (array of integer)

int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
char **array_of_argv[], int array_of_maxprocs[],
MPI_Info array_of_info[], int root, MPI_Comm comm,
MPI_Comm *intercomm, int array_of_errcodes[])

Advice to users. MPI_COMM_GET_PARENT returns a handle to a single intercommunicator. Calling MPI_COMM_GET_PARENT a second time returns a handle to the same intercommunicator. Freeing the handle with MPI_COMM_DISCONNECT or MPI_COMM_FREE will cause other references to the intercommunicator to become invalid (dangling). Note that calling MPI_COMM_FREE on the parent communicator is not useful. (End of advice to users.)

Rationale. 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. (End of rationale.)

10.3.3 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.


IN count number of commands (positive integer, significant to MPI only at root — see advice to users)
IN array_of_commands programs to be executed (array of strings, significant only at root)
IN array_of_argv arguments for commands (array of arrays of strings, significant only at root)
IN array_of_maxprocs maximum number of processes to start for each command (array of integer, significant only at root)
IN array_of_info info objects telling the runtime system where and how to start processes (array of handles, significant only at root)
IN root rank of process in which previous arguments are examined (integer)
IN comm intracommunicator containing group of spawning processes (handle)
OUT intercomm intercommunicator between original group and newly spawned group (handle)
OUT array_of_errcodes one error code per process (array of integer)

int MPI_Comm_spawn_multiple(int count, char *array_of_commands[],
char **array_of_argv[], int array_of_maxprocs[],
MPI_Info array_of_info[], int root, MPI_Comm comm,
MPI_Comm *intercomm, int array_of_errcodes[])
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OFARGV,
ARRAY_OF_MAXPROCS, ARRAY_OFINFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES, ERROR)

INTEGER COUNT, ARRAY_OFINFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
INTERCOMM, ARRAY_OF_ERRCODES(*), ERROR
CHARACTER(*) ARRAY_OF_COMMANDS(*), ARRAY_OFARGV(COUNT, *)

MPI::Intercomm MPI::Intracomm::Spawn_multiple(int count,
const char* array_of_commands[], const char** array_of_argv[],
const int array_of_maxprocs[], const MPI::Info array_of_info[], int root)

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. (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 **)NULL 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_ARGVS_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).

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 m1 processes are generated by the first
command, m2 by the second, etc. The processes corresponding to the first command have
ranks 0, 1, ..., m1-1. The processes in the second command have ranks m1, m1+1, ..., m1+m2-1.
The processes in the third have ranks m1 + m2, m1+m2+1, ..., m1+m2+m3-1, etc.
10.3 PROCESS MANAGER INTERFACE

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, so the two methods are not completely equivalent. There are also two performance-related 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 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 \( \sum_{i=1}^{\text{maxprocs}} n_i \), 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 \( \sum_{j=1}^{i} n_j - 1 \). Error codes are treated as for MPI_COMM_SPAWN.

Example 10.2 Examples of array of argv in C and Fortran

To run the program “ocean” with arguments “gridfile” and “ocean1.grd” and the program “atmos” with argument “atmos.grd” in C:

```c
char *array_of_commands[2] = {"ocean", "atmos"};
char **array_of_argv[2];
char *argv0[3] = {"-gridfile", "ocean1.grd", "atmos.grd", "-gridfile", "ocean1.grd", "atmos.grd", ... = argv0;
array_of_argv[1] = argv1;
MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
```

Here’s how you do it in Fortran:

```fortran
CHARACTER*25 commands(2), array_of_argv(2, 3)
commands(1) = 'ocean'
array_of_argv(1, 1) = 'gridfile'
array_of_argv(2, 1) = 'ocean1.grd'
array_of_argv(2, 2) = 'atmos.grd'
call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)```

10.3.4 Reserved Keys

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.

Example 10.2 Examples of array of argv in C and Fortran

To run the program “ocean” with arguments “gridfile” and “ocean1.grd” and the program “atmos” with argument “atmos.grd” in C:

```c
char *array_of_commands[2] = {"ocean", "atmos"};
char **array_of_argv[2];
char *argv0[3] = {"-gridfile", "ocean1.grd", "atmos.grd", ... = argv0;
array_of_argv[1] = argv1;
MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, ...);
```

Here’s how you do it in Fortran:

```fortran
CHARACTER*25 commands(2), array_of_argv(2, 3)
commands(1) = 'ocean'
array_of_argv(1, 1) = 'gridfile'
array_of_argv(2, 1) = 'ocean1.grd'
array_of_argv(2, 2) = 'atmos.grd'
call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)```
Value is the name of a directory on a machine on which the spawned process(es) execute(s). This directory is made the working directory of the executing process(es). The format of the directory name is determined by the implementation.

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.

dir Value is the name of a directory in which additional information is specified. The format of the filename and internal format of the file are determined by the implementation.

soft Value specifies a set of numbers which are allowed values for the number of processes that MPI_COMM_SPAWN (et al.) may create. The format of the value is a comma-separated 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, ..., a + ck where for c > 0, k is the largest integer for which a + ck ≤ b and for c < 0, k is the largest integer for which a + ck ≥ 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 power-of-two number of processes.
4. 2:10000:2 allows even number of processes.
5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.

10.3.5 Spawn Example
Manager-worker Example, Using MPI_COMM_SPAWN.

```c
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
    int world_size, universe_size, *universe_sizep, flag;
    MPI_Comm everyone; /* intercommunicator */
    char worker_program[100];
    MPI_Comm everyone; /* intercommunicator */
    char worker_program[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```
if (world_size != 1) error("Top heavy with management");

MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
    &universe_sizep, &flag);
if (!flag) {
    printf("This MPI does not support UNIVERSE_SIZE. How many\nprocesses total?");
    scanf("%d", &universe_size);
} else universe_size = *universe_sizep;
if (universe_size == 1) error("No room to start workers");

/*
 * Now spawn the workers. Note that there is a run-time determination
 * of what type of worker to spawn, and presumably this calculation must
 * be done at run time and cannot be calculated before starting
 * the program. If everything is known when the application is
 * first started, it is generally better to start them all at once
 * in a single MPI_COMM_WORLD.
 */

choose_worker_program(worker_program);
MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
    MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone, MPI_ERRCODES_IGNORE);

/*
 * Parallel code here. The communicator "everyone" can be used
 * to communicate with the spawned processes, which have ranks 0...
 * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
 * "everyone".
 */

MPI_Finalize();
return 0;

/* worker */
#include "mpi.h"
int main(int argc, char *argv[])
{
    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");

    MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
        &universe_sizep, &flag);
    if (!flag) {
        printf("This MPI does not support UNIVERSE_SIZE. How many\nprocesses total?");
        scanf("%d", &universe_size);
    } else universe_size = *universe_sizep;
    if (universe_size == 1) error("No room to start workers");

    /*
    * Now spawn the workers. Note that there is a run-time determination
    * of what type of worker to spawn, and presumably this calculation must
    * be done at run time and cannot be calculated before starting
    * the program. If everything is known when the application is
    * first started, it is generally better to start them all at once
    * in a single MPI_COMM_WORLD.
    */

    choose_worker_program(worker_program);
    MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
        MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone, MPI_ERRCODES_IGNORE);

    /*
    * Parallel code here. The communicator "everyone" can be used
    * to communicate with the spawned processes, which have ranks 0...
    * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
    * "everyone".
    */

    MPI_Finalize();
    return 0;
}

/* worker */
#include "mpi.h"
int main(int argc, char *argv[])
{
    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");
10.4 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 intercommunicator, where the two groups of the intercommunicator 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, MPI does not guarantee the traditional robustness of client/server systems. The functionality described in this section is intended to allow two cooperating parts of the same application to communicate with one another. For instance, a client that gets a segmentation fault and dies, or one that doesn’t participate in a collective operation may cause a server to crash or hang. (End of advice to users.)

10.4.1 Names, Addresses, Ports, and All That

Almost all of the complexity in MPI client/server routines addresses the question “how does the client find out how to contact the server?” The difficulty, of course, is that there...
is no existing communication channel between them, yet they must somehow agree on a rendezvous point where they will establish communication — Catch-22.

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 doesn’t really care what server it contacts, only that it be able to get in touch with one that can handle its request.

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 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.

An MPI implementation may allow the server to publish a (port name, service name) pair with MPI_PUBLISH_NAME and the client to retrieve the port name from the service name with MPI_LOOKUP_NAME. This allows three levels of portability, with increasing levels of functionality.

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.
3. Applications may ignore MPI’s name publishing functionality and use their own mechanism (possibly system-supplied) to publish names. This allows arbitrary flexibility but is not portable.
A server makes itself available with two routines. First it must call MPI\_OPEN\_PORT to establish a port at which it may be contacted. Secondly it must call MPI\_COMM\_ACCEPT to accept connections from clients.

`MPI\_OPEN\_PORT(info, port_name)`

`IN` info implementation-specific information on how to establish an address (handle)

`OUT` port_name newly established port (string)

```
int MPI\_Open\_port(MPI\_Info info, char *port\_name)

MPI\_OPEN\_PORT(INFO, PORT\_NAME, IERROR)
```

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`.

**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. (*End of advice to users.*)

`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.

**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 an IP address. A port name may be reused after it is freed with MPI\_CLOSE\_PORT and released by the system.

**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 advice to implementors.*)

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`.

**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. (*End of advice to users.*)

`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.

**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 an IP address. A port name may be reused after it is freed with MPI\_CLOSE\_PORT and released by the system.

**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 advice to implementors.*)
10.4. ESTABLISHING COMMUNICATION

info may be used to tell the implementation how to establish the address. It may, and usually will, be MPI_INFO_NULL in order to get the implementation defaults.

MPI_CLOSE_PORT(port_name)
  IN port_name a port (string)

int MPI_Close_port(char *port_name)
MPI_CLOSE_PORT(PORT_NAME, IERROR)

void MPI::Close_port(const char* port_name)

This function releases the network address represented by port_name.

MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)
  IN port_name port name (string, used only on root)
  IN info implementation-dependent information (handle, used only on root)
  IN root rank in comm of root node (integer)
  IN comm intracommunicator over which call is collective (handle)
  OUT newcomm intercommunicator with client as remote group (handle)

int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)

MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)

MPI::Intercomm MPI::Intracomm::Accept(const char* port_name, const MPI::Info& info, int root) const

MPI_COMM_ACCEPT establishes communication with a client. It is collective over the calling communicator. It returns an intercommunicator that allows communication with the client.

The port_name must have been established through a call to MPI_OPEN_PORT.
info is a implementation-defined string that may allow fine control over the ACCEPT call.

10.4.3 Client Routines

There is only one routine on the client side.
MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)

### Parameters

- **port_name**: network address (string, used only on root)
- **info**: implementation-dependent information (handle, used only on root)
- **root**: rank in comm of root node (integer)
- **comm**: intracommunicator over which call is collective (handle)
- **newcomm**: intercommunicator with server as remote group (handle)

### Returns

- **newcomm**: intercommunicator with server as remote group (handle)

### Description

This routine establishes communication with a server specified by `port_name`. It is collective over the calling communicator and returns an intercommunicator 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`.

**Advice to implementors.** The time out period may be arbitrarily short or long. 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 `MPI::Intracomm`::Connect(const char* port_name, const MPI::Info& info, int root) method, 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.
10.4.4 Name Publishing

The routines in this section provide a mechanism for publishing names. A (service_name, port_name) pair is published by the server, and may be retrieved by a client using the service_name only. An MPI implementation defines the scope of the service_name, that is, the domain over which the service_name can be retrieved. If the domain is the empty set, that is, if no client can retrieve the information, then we say that name publishing is not supported. Implementations should document how the scope is determined. High-quality implementations will give some control to users through the info arguments to name publishing functions. Examples are given in the descriptions of individual functions.

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)

int MPI_Publish_name(const char* service_name, MPI_Info info, char* port_name)

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 name 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 deleted 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 can 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.)

```c
int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)

MPI_UNPUBLISH_NAME(service_name, info, port_name)
IN service_name a service name (string)
IN info implementation-specific information (handle)
IN port_name a port name (string)

This routine unpublishes a service name that has been previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous and is indicated by the error class MPI_ERR_SERVICE.

All published names must be unpublished before the corresponding port is closed and before the publishing process exits. The behavior of MPI_UNPUBLISH_NAME is implementation dependent when a process tries to unpublish a name that it did not publish.

If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, the implementation may require that info passed to MPI_UNPUBLISH_NAME contain information to tell the implementation how to unpublish a name.

MPI_LOOKUP_NAME(service_name, info, port_name)

int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)

MPI_LOOKUP_NAME(service_name, info, port_name)
IN service_name a service name (string)
IN info implementation-specific information (handle)
OUT port_name a port name (string)

This routine unpublishes a service name that has been previously published. Attempting to unpublish a name that has not been published or has already been unpublished is erroneous and is indicated by the error class MPI_ERR_SERVICE.

All published names must be unpublished before the corresponding port is closed and before the publishing process exits. The behavior of MPI_UNPUBLISH_NAME is implementation dependent when a process tries to unpublish a name that it did not publish.

If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, the implementation may require that info passed to MPI_UNPUBLISH_NAME contain information to tell the implementation how to unpublish a name.
```
10.4. Establishing Communication

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.

10.4.5 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).

10.4.6 Client/Server Examples

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:

```c
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:

```c
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
/* do something with intercomm */
```

On the client side:

```c
void MPI::Lookup_name(const char* service_name, const MPI::Info& info, char* port_name)

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.

10.4.5 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).

10.4.6 Client/Server Examples

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:

```c
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:

```c
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
/* do something with intercomm */
```

On the client side:
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);

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);

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_Finalize( &argc, &argv );

}
Here is the client.

```c
#include "mpi.h"
int main( int argc, char **argv )
{
    MPI_Comm server;
    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_Finalize();
    return 0;
}
```

Here is the server.

```c
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 );
}
```
CHAPTER 10. PROCESS CREATION AND MANAGEMENT

10.5 Other Functionality

10.5.1 Universe Size

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 a user (or possibly a batch system) runs one of these quasi-static applications, she 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.

MPI provides an attribute on MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, that allows the application to obtain this information in a portable manner. This attribute indicates the total number of processes that are expected. In Fortran, the attribute is the integer value. In C, the attribute is a pointer to the integer value. An application typically subtracts the size of MPI_COMM_WORLD from MPI_UNIVERSE_SIZE to find out how many processes it should spawn. MPI_UNIVERSE_SIZE is initialized in MPI_INIT and is not changed by MPI. If defined, it has the same value on all processes of MPI_COMM_WORLD. MPI_UNIVERSE_SIZE is determined by the application startup mechanism in a way not specified by MPI. (The size of MPI_COMM_WORLD is another example of such a parameter.)

Possibilities for how MPI_UNIVERSE_SIZE might be set include
- A \texttt{-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 \texttt{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.

MPI_UNIVERSE_SIZE is a recommendation, not necessarily a hard limit. For instance, some implementations may allow an application to spawn 50 processes per processor, if they are requested. However, it is likely that the user only wants to spawn one process per processor.

MPI_UNIVERSE_SIZE is assumed to have been specified when an application was started, and is in essence a portable mechanism to allow the user to pass to the application (through the MPI process startup mechanism, such as \texttt{mpiexec}) a piece of critical runtime information. Note that no interaction with the runtime environment is required. If the runtime environment changes size while an application is running, MPI_UNIVERSE_SIZE is not updated, and the application must find out about the change through direct communication with the runtime system.

10.5.2 Singleton MPI_INIT

A high-quality implementation will allow any process (including those not started with a "parallel application" mechanism) to become an MPI process by calling MPI_INIT. Such a process can then connect to other MPI processes using the MPI_COMM_ACCEPT and
10.5. OTHER FUNCTIONALITY

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. To start MPI processes belonging to the same MPI_COMM_WORLD requires some special coordination. 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.

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.)

10.5.3 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 implementation-specific 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

```bash
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 doesn’t 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.

Advice to implementors. To start MPI processes belonging to the same MPI_COMM_WORLD requires some special coordination. 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.

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.)

10.5.3 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 implementation-specific 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

```bash
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 doesn’t 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.
\[\text{appnum} \text{ Value contains an integer that overrides the default value for } \text{MPI_APPNUM} \text{ in the child.}\]

\textit{Rationale.} When a single application is started, it is able to figure out how many processes there are by looking at the size of \texttt{MPI_COMM_WORLD}. An application consisting of multiple \texttt{MPI} 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. \texttt{MPI_APPNUM} provides such a general mechanism. (End of rationale.)

10.5.4 Releasing Connections

Before a client and server connect, they are independent \texttt{MPI} applications. An error in one does not affect the other. After establishing a connection with \texttt{MPI_COMM_CONNECT} and \texttt{MPI_COMM_ACCEPT}, an error in one may affect the other. It is desirable for a client and server to be able to disconnect, so that an error in one will not affect the other. Similarly, it might be desirable for a parent and child to disconnect, so that errors in the child do not affect the parent, or vice-versa.

- 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 \texttt{MPI_COMM_WORLD}) or
     b. they have previously belonged to a communicator that was freed with \texttt{MPI_COMM_FREE} instead of \texttt{MPI_COMM_DISCONNECT} or
     c. they both belong to the group of the same window or filehandle.
  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 \texttt{MPI} processes into disconnected (independent) sets (equivalence classes) of processes.

- Processes which are connected, but don’t share the same \texttt{MPI_COMM_WORLD} may become disconnected (independent) if the communication path between them is broken by using \texttt{MPI_COMM_DISCONNECT}.

The following additional rules apply to \texttt{MPI} routines in other chapters:

- \texttt{MPI_FINALIZE} is collective over a set of connected processes.
- \texttt{MPI_ABORT} does not abort independent processes. It may abort all processes in the caller’s \texttt{MPI_COMM_WORLD} (ignoring its \texttt{comm} argument). Additionally, it may abort connected processes as well, though it makes a “best attempt” to abort only the processes in \texttt{comm}.
- If a process terminates without calling \texttt{MPI_FINALIZE}, independent processes are not affected but the effect on connected processes is not defined.

\[\text{appnum} \text{ Value contains an integer that overrides the default value for } \text{MPI_APPNUM} \text{ in the child.}\]

\textit{Rationale.} When a single application is started, it is able to figure out how many processes there are by looking at the size of \texttt{MPI_COMM_WORLD}. An application consisting of multiple \texttt{MPI} 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. \texttt{MPI_APPNUM} provides such a general mechanism. (End of rationale.)

10.5.4 Releasing Connections

Before a client and server connect, they are independent \texttt{MPI} applications. An error in one does not affect the other. After establishing a connection with \texttt{MPI_COMM_CONNECT} and \texttt{MPI_COMM_ACCEPT}, an error in one may affect the other. It is desirable for a client and server to be able to disconnect, so that an error in one will not affect the other. Similarly, it might be desirable for a parent and child to disconnect, so that errors in the child do not affect the parent, or vice-versa.

- 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 \texttt{MPI_COMM_WORLD}) or
     b. they have previously belonged to a communicator that was freed with \texttt{MPI_COMM_FREE} instead of \texttt{MPI_COMM_DISCONNECT} or
     c. they both belong to the group of the same window or filehandle.
  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 \texttt{MPI} processes into disconnected (independent) sets (equivalence classes) of processes.

- Processes which are connected, but don’t share the same \texttt{MPI_COMM_WORLD} may become disconnected (independent) if the communication path between them is broken by using \texttt{MPI_COMM_DISCONNECT}.

The following additional rules apply to \texttt{MPI} routines in other chapters:

- \texttt{MPI_FINALIZE} is collective over a set of connected processes.
- \texttt{MPI_ABORT} does not abort independent processes. It may abort all processes in the caller’s \texttt{MPI_COMM_WORLD} (ignoring its \texttt{comm} argument). Additionally, it may abort connected processes as well, though it makes a “best attempt” to abort only the processes in \texttt{comm}.
- If a process terminates without calling \texttt{MPI_FINALIZE}, independent processes are not affected but the effect on connected processes is not defined.
10.5. OTHER FUNCTIONALITY

MPI_COMM_DISCONNECT(comm)
INOUT comm communicator (handle)

int MPI_Comm_disconnect(MPI_Comm *comm)

MPI_COMM_DISCONNECT(COMM, IERROR)
INTEGER COMM, IERROR

void MPI::Comm::Disconnect()

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. Notes 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.)

10.5.5 Another Way to Establish MPI Communication

MPI_COMM_JOIN(fd, intercomm)
IN fd socket file descriptor
OUT intercomm new intercommunicator (handle)

int MPI_Comm_join(int fd, MPI_Comm *intercomm)

MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
INTEGER FD, INTERCOMM, IERROR

static MPI::Intercomm MPI::Comm::Join(const int fd)

MPI_COMM_JOIN is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [33, 37]. Implementations that exist in an environment not supporting Berkeley Sockets should provide the entry point for MPI_COMM_JOIN and should return MPI_COMM_NULL.

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. Notes 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.)
This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI_COMMJOIN should normally succeed if the local and remote processes have access to the same implementation-defined MPI communication universe.

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (End of advice to users.)

Advice to implementors. A high-quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (End of advice to implementors.)

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable byte-stream connection). Non-blocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI_COMMJOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls. MPI_COMMJOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI_COMMJOIN. The two processes are referred to as the local and remote processes. MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing else. Upon return from MPI_COMMJOIN, the file descriptor will be open and quiescent (see below).

If MPI is unable to create an intercommunicator, 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_COMMJOIN is called and after MPI_COMMJOIN returns. More specifically, on entry to MPI_COMMJOIN, a read on the socket will not read any data that was written to the socket before the remote process called MPI_COMMJOIN. On exit from MPI_COMMJOIN, a read will not read any data that was written to the socket before the remote process returned from MPI_COMMJOIN. 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_COMMJOIN, or call MPI_COMMJOIN 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_COMMJOIN 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_COMMJOIN on two connected processes (see Section 10.5.4 on page 318 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.

This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI_COMMJOIN should normally succeed if the local and remote processes have access to the same implementation-defined MPI communication universe.

Advice to users. An MPI implementation may require a specific communication medium for MPI communication, such as a shared memory segment or a special switch. In this case, it may not be possible for two processes to successfully join even if there is a socket connecting them and they are using the same MPI implementation. (End of advice to users.)

Advice to implementors. A high-quality implementation will attempt to establish communication over a slow medium if its preferred one is not available. If implementations do not do this, they must document why they cannot do MPI communication over the medium used by the socket (especially if the socket is a TCP connection). (End of advice to implementors.)

fd is a file descriptor representing a socket of type SOCK_STREAM (a two-way reliable byte-stream connection). Non-blocking I/O and asynchronous notification via SIGIO must not be enabled for the socket. The socket must be in a connected state. The socket must be quiescent when MPI_COMMJOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls. MPI_COMMJOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI_COMMJOIN. The two processes are referred to as the local and remote processes. MPI uses the socket to bootstrap creation of the intercommunicator, and for nothing else. Upon return from MPI_COMMJOIN, the file descriptor will be open and quiescent (see below).

If MPI is unable to create an intercommunicator, 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_COMMJOIN is called and after MPI_COMMJOIN returns. More specifically, on entry to MPI_COMMJOIN, a read on the socket will not read any data that was written to the socket before the remote process called MPI_COMMJOIN. On exit from MPI_COMMJOIN, a read will not read any data that was written to the socket before the remote process returned from MPI_COMMJOIN. 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_COMMJOIN, or call MPI_COMMJOIN 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_COMMJOIN 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_COMMJOIN on two connected processes (see Section 10.5.4 on page 318 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.
Chapter 11

One-Sided Communications

11.1 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 update at other processes. However, processes may not know which data in their own memory need to be accessed or updated by remote processes, and may not even know the identity of these processes. 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 may require all processes to participate in a time consuming global computation, or to periodically poll for potential communication requests to receive and act upon. 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 \( \text{MPI A = B(map)} \), where \( \text{map} \) is a permutation vector, and \( A, B \) and \( \text{map} \) are distributed in the same manner.

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. Three communication calls are provided: MPI_PUT (remote write), MPI_GET (remote read) and MPI_ACCUMULATE (remote update). A larger number of synchronization calls are provided that support different synchronization styles. The design is similar to that of weakly coherent memory systems: correct ordering of memory accesses has to be imposed by the user, using synchronization calls; the implementation can delay communication operations until the synchronization calls occur, for efficiency.

The design of the RMA functions allows implementors to take advantage, in many cases, of fast communication mechanisms provided by various platforms, such as coherent or noncoherent shared memory, DMA engines, hardware-supported put/get operations, communication coprocessors, etc. The most frequently used RMA communication mechanisms can be layered on top of message-passing. However, support for asynchronous communication agents (handlers, threads, etc.) is needed, for certain RMA functions, in a distributed memory environment.

We shall denote by \( \text{origin} \) the process that performs the call, and by \( \text{target} \) the process that performs the call, and by \( \text{origin} \) the
process in which the memory is accessed. Thus, in a put operation, source=origin and destination=target; in a get operation, source=target and destination=origin.

### 11.2 Initialization

#### 11.2.1 Window Creation

The initialization operation allows each process in an intracommunicator group to specify, in a collective operation, 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.

```c
MPI_Win_create(base, size, disp_unit, info, comm, win)
```

**Parameters:**
- `base`: initial address of window (choice)
- `size`: size of window in bytes (nonnegative integer)
- `disp_unit`: local unit size for displacements, in bytes (positive integer)
- `info`: info argument (handle)
- `comm`: communicator (handle)
- `win`: window object returned by the call (handle)

This is a collective call executed by all processes in the group of `comm`. It returns a window object that can be used by these processes to perform RMA operations. Each process 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 `base`. A process may elect to expose no memory by specifying `size = 0`.

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 `disp_unit` specified by the target process, at window creation.

**Rationale.** The window size is specified using an address sized integer, so as to allow windows that span more than 4 GB of address space. (Even if the physical memory size is less than 4 GB, the address range may be larger than 4 GB, if addresses are not contiguous.) *(End of rationale.)*
Advice to users. Common choices for disp_unit are 1 (no scaling), and (in C syntax) sizeof(type), for a window that consists of an array of elements of type type. The later choice will allow one to use array indices in RMA calls, and have those scaled correctly to byte displacements, even in a heterogeneous environment. (End of advice to users.)

The info argument provides optimization hints to the runtime about the expected usage pattern of the window. The following info key is predefined:

no_locks — if set to true, then the implementation may assume that the local window is never locked (by a call to MPI_WIN_LOCK). This implies that this window is not used for 3-party communication, and RMA can be implemented with no (less) asynchronous agent activity at this process.

The various processes in the group of comm may specify completely different target windows, in location, size, displacement units and info arguments. As long as all the get, 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 erroneous results.

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 8.2, page 262) 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 allow “good” memory to be used for static variables.

Implementors should document any performance impact of window alignment. (End of advice to implementors.)

MPI_WIN_FREE(win)
INOUT win window object (handle)

int MPI_Win_free(MPI_Win *win)

MPI_WIN_FREE(WIN, IERROR)
### 11.2.2 Window Attributes

The following three attributes are cached with a window, when the window is created.

- **MPI_WIN_BASE** — window base address.
- **MPI_WIN_SIZE** — window size, in bytes.
- **MPI_WIN_DISP_UNIT** — displacement unit associated with the window.

In C, to call `MPI_Win_get_attr`:

```c
MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag);
MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag);
MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag);
```

In Fortran, to call `MPI_Win_Get_Attr`:

```fortran
MPI_WIN_GETATTR(WIN, BASE, base, flag, ierror);
MPI_WIN_GETATTR(WIN, SIZE, size, flag, ierror);
MPI_WIN_GETATTR(WIN, DISP_UNIT, disp_unit, flag, ierror);
```

The other "window attribute," namely the group of processes attached to the window, can be retrieved using the call below.

```c
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
```

```fortran
int MPI_WIN_GETGROUP(WIN, GROUP, IERROR)
```

Advice to implementors. **MPI_WIN_FREE** requires a barrier synchronization: no process can return from free until all processes in the group of win called free. This, to ensure that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. (End of advice to implementors.)

---

### 11.2.3 Window Attributes

The other "window attribute," namely the group of processes attached to the window, can be retrieved using the call below.

```c
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
```

```fortran
int MPI_WIN_GETGROUP(WIN, GROUP, IERROR)
```
MPI provides an RMA operation to move data in its memory.

For all three calls, the target process may be identical with the origin process; i.e., a
provided by the caller.

It at the destination. However, all arguments specifying both communication buffers are
and at the target. Thus, a transfer operation may also gather data at the source and scatter

The calls use general datatype arguments to specify communication buffers at the origin
page 349.

RMA locations in the window. The last restriction enables more efficient implementations of
operation and by a local store operation. This, even if the two updates access different

It is erroneous to have concurrent conflicting accesses to the same memory location in a
window; if a location is updated by a put or accumulate operation, then this location cannot
be accessed by a load or another RMA operation until the updating operation has completed
in some order. In addition, a window cannot concurrently be updated by a put or accumulate
operation and by a local store operation. This, even if these two updates access different
locations in the window. The last restriction enables more efficient implementations of RMA
operations on many systems. These restrictions are described in more detail in Section 11.7,
page 349.

The calls use general datatype arguments to specify communication buffers at the origin
and at the target. Thus, a transfer operation may also gather data at the source and scatter
it at the destination. However, all arguments specifying both communication buffers are
provided by the caller.

For all three calls, the target process may be identical with the origin process; i.e., a
process may use an RMA operation to move data in its memory.
CHAPTER 11. ONE-SIDED COMMUNICATIONS

Rationale. The choice of supporting “self-communication” is the same as for message-passing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (End of rationale.)

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. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch.

11.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

IN origin_addr
initial address of origin buffer (choice)

IN origin_count
number of entries in origin buffer (nonnegative integer)

IN origin_datatype
datatype of each entry in origin buffer (handle)

IN target_rank
rank of target (nonnegative integer)

IN target_disp
displacement from start of window to target buffer (nonnegative integer)

IN target_count
number of entries in target buffer (nonnegative integer)

IN target_datatype
datatype of each entry in target buffer (handle)

IN win
window object used for communication (handle)

int MPI_Put(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_Win win)

MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)

void MPI::Win::Put(const void* origin_addr, int origin_count, const MPI::Datatype& origin_datatype, int target_rank, MPI::Aint target_disp, int target_count, const MPI::Datatype& target_datatype) const

Rationale. The choice of supporting “self-communication” is the same as for message-passing. It simplifies some coding, and is very useful with accumulate operations, to allow atomic updates of local variables. (End of rationale.)

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. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch.

11.3.1 Put

The execution of a put operation is similar to the execution of a send by the origin process and a matching receive by the target process. The obvious difference is that all arguments are provided by one call — the call executed by the origin process.

MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)

IN origin_addr
initial address of origin buffer (choice)

IN origin_count
number of entries in origin buffer (nonnegative integer)

IN origin_datatype
datatype of each entry in origin buffer (handle)

IN target_rank
rank of target (nonnegative integer)

IN target_datatype
datatype of each entry in target buffer (handle)

IN win
window object used for communication (handle)

int MPI_Put(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_Win win)

MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)

void MPI::Win::Put(const void* origin_addr, int origin_count, const MPI::Datatype& origin_datatype, int target_rank, MPI::Aint target_disp, int target_count, const MPI::Datatype& target_datatype) const
11.3. COMMUNICATION CALLS

Transfers \texttt{origin\_count} successive entries of the type specified by the \texttt{origin\_datatype}, starting at address \texttt{origin\_addr} on the origin node to the target node specified by the \texttt{win\_target\_rank} pair. The data are written in the target buffer at address \texttt{target\_addr = window\_base + target\_disp \times disp\_unit}, where \texttt{window\_base} and \texttt{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 \texttt{target\_count} and \texttt{target\_datatype}.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments \texttt{origin\_addr, origin\_count, origin\_datatype, target\_rank, tag, comm}, and the target process executed a receive operation with arguments \texttt{target\_addr, target\_count, target\_datatype, source, tag, comm}, where \texttt{target\_addr} is the target buffer address computed as explained above, and \texttt{comm} is a communicator for the group of windows.

The communication must satisfy the same constraints as for a similar message-passing communication. The \texttt{target\_datatype} may not specify overlapping entries in the target buffer. The message sent must fit, without truncation, in the target buffer. Furthermore, the target buffer must fit in the target window.

The \texttt{target\_datatype} argument is a handle to a datatype object defined at the origin process. However, this object is interpreted at the target process: the outcome is as if the \texttt{target\_datatype} object was defined at the target process, by the same sequence of calls used to define it at the origin process. The \texttt{target\_datatype} must contain only relative displacements, not absolute addresses. The same holds for get and accumulate.

\textit{Advice to users.} The \texttt{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, page 11).

The performance of a put transfer can be significantly affected, on some systems, from the choice of window location and the shape and location of the origin and target buffer: transfers to a target window in memory allocated by \texttt{MPI\_ALLOC\_MEM} may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (End of advice to users.)

\textit{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, both for debugging purposes, and for protection with client-server codes that use RMA. I.e., a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurred. 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.)
11.3.2 Get

\[
\text{MPI\_Get(origin\_addr, origin\_count, origin\_datatype, target\_rank, target\_disp, target\_count, target\_datatype, win)}
\]

**OUT** origin\_addr
initial address of buffer (choice)

**IN** origin\_count
number of entries in origin buffer (nonnegative integer)

**IN** origin\_datatype
datatype of each entry in origin buffer (handle)

**IN** target\_rank
rank of target (nonnegative integer)

**IN** target\_disp
 displacement from window start to the beginning of the target buffer (nonnegative integer)

**IN** target\_count
number of entries in target buffer (nonnegative integer)

**IN** target\_datatype
datatype of each entry in target buffer (handle)

**IN** win
window object used for communication (handle)

```
int MPI\_Get(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\_Win win)
```

Similar to \texttt{MPI\_PUT}, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The \texttt{origin\_datatype} may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window, and the copied data must fit, without truncation, in the origin buffer.

11.3.3 Examples

**Example 11.1** We show how to implement the generic indirect assignment \( A = B(\text{map}) \), where \( A \), \( B \), and \( \text{map} \) have the same distribution, and \( \text{map} \) is a permutation. To simplify, we assume a block distribution with equal size blocks.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
```

Similar to \texttt{MPI\_PUT}, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The \texttt{origin\_datatype} may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window, and the copied data must fit, without truncation, in the origin buffer.

**Example 11.1** We show how to implement the generic indirect assignment \( A = B(\text{map}) \), where \( A \), \( B \), and \( \text{map} \) have the same distribution, and \( \text{map} \) is a permutation. To simplify, we assume a block distribution with equal size blocks.
REAL A(m), B(m)

INTEGER otype(p), oindex(m), & ! used to construct origin datatypes
ttype(p), tindex(m), & ! used to construct target datatypes
count(p), total(p), & win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal

! This part does the work that depends on the locations of B.
! Can be reused while this does not change
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL, &
  comm, win, ierr)

! 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)) = i
Example 11.2 A simpler version can be written that does not require that a datatype be built for the target buffer. But, one then needs a separate get call for each entry, as illustrated below. This code is much simpler, but usually much less efficient, for large arrays.

```
SUBROUTINE MAPVALS(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p
REAL A(m), B(m)
INTEGER win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL, &
comm, win, ierr)
DO i=1,m
  j = map(i)/pk = MOD(map(i),p)
  CALL MPI_Type_create_indexed_block(count(i), 1, tindex(total(j)+count(j)), &
MPI_REAL, ttype(i), ierr)
  CALL MPI_Type_commit(ttype(i), ierr)
END DO
CALL MPI_WIN_fence(0, win, ierr)
DO i=1,m
  CALL MPI_Type_free(ttype(i), ierr)
END DO
RETURN
END
```
11.3. COMMUNICATION CALLS

CALL MPI_GET(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
RETURN
END

11.3.4 Accumulate Functions

It is often useful in a put operation to combine the data moved to the target process with the data that resides at that process, rather than replacing the data there. This will allow, for example, the accumulation of a sum by having all involved processes add their contribution to the sum variable in the memory of one process.

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)
IN origin_count number of entries in buffer (nonnegative integer)
IN origin_datatype datatype of each buffer entry (handle)
IN target_rank rank of target (nonnegative integer)
IN target_disp displacement from start of window to beginning of target buffer (nonnegative integer)
IN target_count number of entries in target buffer (nonnegative integer)
IN target_datatype datatype of each entry in target buffer (handle)
IN op reduce operation (handle)
IN win window object (handle)

int MPI_Accumulate(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)

MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
Accumulate the contents of the origin buffer (as defined by `origin_addr`, `origin_count` and `origin_datatype`) to the buffer specified by arguments `target_count` and `target_datatype`, at offset `target_disp`, in the target window specified by `target_rank` and `win`, using the operation `op`. This is like `MPI_PUT` except that data is combined into the target area instead of overwriting it.

Any of the predefined operations for `MPI_REDUCE` can be used. User-defined functions cannot be used. For example, if `op` is `MPI_SUM`, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both 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.

A new predefined operation, `MPI_REPLACE`, is defined. It corresponds to the associative function \( f(a, b) = b \); i.e., the current value in the target memory is replaced by the value supplied by the origin. `MPI_REPLACE`, like the other predefined operations, is defined only for the predefined MPI datatypes.

**Rationale.** The rationale for this is that, for consistency, `MPI_REPLACE` should have the same limitations as the other operations. Extending it to all datatypes doesn’t provide any real benefit. (End of rationale.)

**Advice to users.** `MPI_PUT` is a special case of `MPI_ACCUMULATE`, with the operation `MPI_REPLACE`. Note, however, that `MPI_PUT` and `MPI_ACCUMULATE` have different constraints on concurrent updates. (End of advice to users.)

**Example 11.3** We want to compute \( B(j) = \sum_{i=1}^{\text{map}(i)} A(i) \). The arrays \( A, B \) and `map` are distributed in the same manner. We write the simple version.

```fortran
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, ierr
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL, &
comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/pk = MOD(map(i),p)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, &
MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
END SUBROUTINE SUM
```

Accumulate the contents of the origin buffer (as defined by `origin_addr`, `origin_count` and `origin_datatype`) to the buffer specified by arguments `target_count` and `target_datatype`, at offset `target_disp`, in the target window specified by `target_rank` and `win`, using the operation `op`. This is like `MPI_PUT` except that data is combined into the target area instead of overwriting it.

Any of the predefined operations for `MPI_REDUCE` can be used. User-defined functions cannot be used. For example, if `op` is `MPI_SUM`, each element of the origin buffer is added to the corresponding element in the target, replacing the former value in the target.

Each datatype argument must be a predefined datatype or a derived datatype, where all basic components are of the same predefined datatype. Both 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.

A new predefined operation, `MPI_REPLACE`, is defined. It corresponds to the associative function \( f(a, b) = b \); i.e., the current value in the target memory is replaced by the value supplied by the origin. `MPI_REPLACE`, like the other predefined operations, is defined only for the predefined MPI datatypes.

**Rationale.** The rationale for this is that, for consistency, `MPI_REPLACE` should have the same limitations as the other operations. Extending it to all datatypes doesn’t provide any real benefit. (End of rationale.)

**Advice to users.** `MPI_PUT` is a special case of `MPI_ACCUMULATE`, with the operation `MPI_REPLACE`. Note, however, that `MPI_PUT` and `MPI_ACCUMULATE` have different constraints on concurrent updates. (End of advice to users.)

**Example 11.3** We want to compute \( B(i) = \sum_{j=1}^{\text{map}(j)} A(j) \). The arrays \( A, B \) and `map` are distributed in the same manner. We write the simple version.

```fortran
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, ierr
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, sizeofreal
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, sizeofreal, ierr)
CALL MPI_WIN_CREATE(B, m*sizeofreal, sizeofreal, MPI_INFO_NULL, &
comm, win, ierr)
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/pk = MOD(map(i),p)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, k, 1, MPI_REAL, &
MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
END SUBROUTINE SUM
```
11.4 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 (`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 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.

----

```fortran
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 11.2, page 330, except that a call to get has been replaced by a call to accumulate. (Note that, if `map` is one-to-one, then 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 11.1, page 328, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

```
CALL MPI_WIN_FREE(win, ierr)
RETURN
END
```

This code is identical to the code in Example 11.2, page 330, except that a call to get has been replaced by a call to accumulate. (Note that, if `map` is one-to-one, then 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 11.1, page 328, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

11.4 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 (`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 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. The MPI_Win_fence collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates 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 MPI_Win_wait can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly DMA accesses to a window with respect to local accesses to that same window. This mechanism may be more efficient when each process communicates with few (logical) neighbors, and the communication graph is fixed or changes infrequently.

   These calls are used for active target communication. An access epoch is started at the origin process by a call to MPI_Win_start and is terminated by a call to MPI_Win_complete. The start call has a group argument that specifies the set of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI_Win_post and is completed by a call to MPI_Win_wait. The post call has a group argument that specifies the set of origin processes for that epoch.

3. Finally, shared and exclusive locks are provided by the two functions MPI_Win_lock and MPI_Win_unlock. Lock synchronization is useful for MPI applications that emulate a shared memory model via MPI calls; e.g., in a "billboard" model, where processes can, at random times, access or update different parts of the billboard.

   These two calls provide passive target communication. An access epoch is started by a call to MPI_Win_lock and terminated by a call to MPI_Win_unlock. Only one target window can be accessed during that epoch with win.

Figure 11.1 illustrates the general synchronization pattern for active target communication. The synchronization between post and start ensures that the put call of the origin process does not start until the target process exposes the window (with the post call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between complete and wait ensures that the put call of the origin process completes before the window is unexposed (with the wait call). The target process will execute following local accesses to the target window only after the wait returned.

Figure 11.1 shows operations occurring in the natural temporal order implied by the synchronizations: the post occurs before the matching start, and complete occurs before the matching wait. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as
11.4. SYNCHRONIZATION CALLS

Figure 11.1: Active target communication. Dashed arrows represent synchronizations (ordering of events).
Figure 11.2: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)
Figure 11.3: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

illustrated in Figure 11.2. 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 11.3 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.
11.4.1 Fence

MPI_WIN_FENCE(assert, win)

Program assertion (integer)

int MPI_Win_fence(int assert, MPI_Win win)

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call 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 the local window was the target of RMA accesses between these two calls. The call completes an RMA exposure epoch if it was preceded by another fence call and the local window was the target of RMA accesses between these two calls. The call starts an RMA access epoch if it is followed by another fence call and by RMA communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and the local window is the target of RMA accesses between these two fence calls.

A fence call usually entails a barrier synchronization: a process completes a call to MPI_WIN_FENCE only after all other processes in the group entered their matching call. However, a call to MPI_WIN_FENCE that is known not to end any epoch (in particular, a call with assert = MPI_MODE_NOPRECEDE) does not necessarily act as a barrier.

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.4.4. A value of assert = 0 is always valid.

Advice to users. Calls to MPI_WIN_FENCE should both precede and follow calls to put, get or accumulate that are synchronized with fence calls. (End of advice to users.)
11.4. SYNCHRONIZATION CALLS

11.4.2 General Active Target Synchronization

MPI\_WIN\_START(group, assert, win)

\[
\begin{align*}
\text{IN} & \quad \text{group} \quad \text{group of target processes (handle)} \\
\text{IN} & \quad \text{assert} \quad \text{program assertion (integer)} \\
\text{IN} & \quad \text{win} \quad \text{window object (handle)} \\
\end{align*}
\]

\[
\text{int MPI\_Win\_start(MPI\_Group group, int assert, MPI\_Win win)}
\]

\[
\text{MPI\_WIN\_START(GROUP, ASSERT, WIN, IERROR)}
\]

\[
\text{INTEGER GROUP, ASSERT, WIN, IERROR}
\]

\[
\text{void MPI::Win::Start(const MPI::Group& group, int assert) const}
\]

Starts an RMA access epoch for \( \text{win} \). RMA calls issued on \( \text{win} \) during this epoch must access only windows at processes in \( \text{group} \). Each process in \( \text{group} \) must issue a matching call to MPI\_WIN\_POST. RMA accesses to each target window will be delayed, if necessary, until the target process executed the matching call to MPI\_WIN\_POST. MPI\_WIN\_START is allowed to block until the corresponding MPI\_WIN\_POST calls are executed, but is not required to.

The \( \text{assert} \) argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.4.4. A value of \( \text{assert} = 0 \) is always valid.

MPI\_WIN\_COMPLETE(win)

\[
\begin{align*}
\text{IN} & \quad \text{win} \quad \text{window object (handle)} \\
\end{align*}
\]

\[
\text{int MPI\_Win\_complete(MPI\_Win win)}
\]

\[
\text{MPI\_WIN\_COMPLETE(WIN, IERROR)}
\]

\[
\text{INTEGER WIN, IERROR}
\]

\[
\text{void MPI::Win::Complete() const}
\]

Completes an RMA access epoch on \( \text{win} \) started by a call to MPI\_WIN\_START. All RMA communication calls issued on \( \text{win} \) 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 11.4

\[
\begin{align*}
\text{MPI\_Win\_start(group, flag, win);} \\
\text{MPI\_Put(\ldots,win);} \\
\text{MPI\_Win\_complete(win);} \\
\end{align*}
\]
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_POST occurred; 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 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)

int MPI_win_post(MPI_Group group, int assert, MPI_Win win)
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)

void MPI::Win::Post() const

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.

MPI_WIN_WAIT(win)

IN win window object (handle)

int MPI_win_wait(MPI_Win win)
MPI_WIN_WAIT(WIN, IERROR)

void MPI::Win::Wait() const

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.

Figure 11.4 illustrates the use of these four functions. Process 0 puts data in the...
11.4. SYNCHRONIZATION CALLS

Figure 11.4: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

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, put or complete calls may occur ahead of the matching post calls.

MPI\_WIN\_TEST(win, flag)

\begin{verbatim}
in MPI\_Win\_test(MPI\_Win win, int *flag)
    
    INTEGER WIN, IERROR
    LOGICAL FLAG

    bool MPI::Win::Test() const
        
        This is the nonblocking version of MPI\_WIN\_WAIT. It returns flag = true if MPI\_WIN\_WAIT would return, flag = false, otherwise. 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 call can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

        MPI\_WIN\_POST(group,0,win) initiate a nonblocking send with tag tag0 to each process in
\end{verbatim}

int MPI\_Win\_test(MPI\_Win win, int *flag)

MPI\_WIN\_TEST(WIN, FLAG, IERROR)

INTEGER WIN, IERROR
LOGICAL FLAG

bool MPI::Win::Test() const

This is the nonblocking version of MPI\_WIN\_WAIT. It returns flag = true if MPI\_WIN\_WAIT would return, flag = false, otherwise. 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 call can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

MPI\_WIN\_POST(group,0,win) initiate a nonblocking send with tag tag0 to each process in
CHAPTER 11. ONE-SIDED COMMUNICATIONS

MPI_WIN_START(group,0,win) initiate 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) initiate 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) initiate 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: 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 \( G = < V, E > \), where \( V = \{0, \ldots, n - 1\} \) 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(outgroup, \ldots), followed by a call to MPI_WIN_START(ingroup, \ldots), where \( outgroup_p = \{ j : ij \in E \} \) and \( ingroup_p = \{ j : ji \in E \} \). 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.)

11.4.3 Lock

MPI_WIN_LOCK(lock_type, rank, assert, win)

IN lock_type either MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED (state)
IN rank rank of locked window (nonnegative integer)
IN assert program assertion (integer)
IN win window object (handle)
11.4. SYNCHRONIZATION CALLS

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
MPI_WIN_LOCK(LOCK_TYPE, RANK, assert, WIN, IERROR)
INTEGER LOCK_TYPE, RANK, assert, WIN, IERROR

void MPI::Win::Lock(int lock_type, int rank, int assert) const

Starts an RMA access epoch. Only the window at the process with rank rank can be accessed by RMA operations on win during that epoch.

MPI_Win_unlock(rank, win)
IN rank rank of window (nonnegative integer)
IN win window object (handle)

int MPI_Win_unlock(int rank, MPI_Win win)
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
INTEGER RANK, WIN, IERROR

void MPI::Win::Unlock(int rank) const

Completes an RMA access epoch started by a call to MPI_WIN_LOCK(...). RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock call, and to protect local load/store accesses to a locked local window executed between the lock and unlock call. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. I.e., a process may not call MPI_WIN_LOCK to lock a target window if the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it is erroneous to call MPI_WIN_POST while the local window is locked.

Rationale. An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage 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 8.2, page 262). Locks can be used portably only in such memory.

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
MPI_WIN_LOCK(LOCK_TYPE, RANK, assert, WIN, IERROR)
INTEGER LOCK_TYPE, RANK, assert, WIN, IERROR

void MPI::Win::Lock(int lock_type, int rank, int assert) const

Starts an RMA access epoch. Only the window at the process with rank rank can be accessed by RMA operations on win during that epoch.

MPI_Win_unlock(rank, win)
IN rank rank of window (nonnegative integer)
IN win window object (handle)

int MPI_Win_unlock(int rank, MPI_Win win)
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
INTEGER RANK, WIN, IERROR

void MPI::Win::Unlock(int rank) const

Completes an RMA access epoch started by a call to MPI_WIN_LOCK(...). RMA operations issued during this period will have completed both at the origin and at the target when the call returns.

Locks are used to protect accesses to the locked target window effected by RMA calls issued between the lock and unlock call, and to protect local load/store accesses to a locked local window executed between the lock and unlock call. Accesses that are protected by an exclusive lock will not be concurrent at the window site with other accesses to the same window that are lock protected. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by an exclusive lock to the same window.

It is erroneous to have a window locked and exposed (in an exposure epoch) concurrently. I.e., a process may not call MPI_WIN_LOCK to lock a target window if the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it is erroneous to call MPI_WIN_POST while the local window is locked.

Rationale. An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage 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 8.2, page 262). Locks can be used portably only in such memory.
Rationale. The implementation of passive target communication when memory is not shared requires an asynchronous 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 3rd party communication in shared memory machines.

The downside of this decision is that passive target communication cannot be used without taking advantage of nonstandard Fortran features: namely, the availability of C-like pointers; these are not supported by some Fortran compilers (g77 and Windows/NT compilers, at the time of writing). Also, passive target communication cannot be portably targeted to COMMON blocks, or other statically declared Fortran arrays. (End of rationale.)

Consider the sequence of calls in the example below.

**Example 11.5**

```
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 call MPI_WIN_LOCK may not block, while the call to MPI_PUT blocks until a lock 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.

11.4.4 Assertions

The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE and MPI_WIN Lock is used to provide assertions on the context of 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 provides 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 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:
MPI_MODE_NOCHECK, MPI_MODE_NOSTORE, MPI_MODE_NOPUT, 
MPI_MODE_NOPRECEDE and MPI_MODE_NOSUCCEED. The significant options are listed 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. Fortran 77 users can use (non-portably) 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.)

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 local 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.

MPI_WIN_FENCE:
MPI_MODE_NOSTORE — the local window was not updated by local 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 this assertion is given by any process in the window group, then it must be given by all processes in the group.

MPI_WIN_LOCK:
CHAPTER 11. ONE-SIDED COMMUNICATIONS

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.)

11.4.5 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as argument 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.

11.5 Examples

Example 11.6 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.


while(!converged(A)){
    update(A);
    MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
    for(i=0; i < toneighbors; i++)
        MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                     todisp[i], 1, totype[i], win);
    MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}

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 11.7 Same generic example, with more computation/communication overlap. We assume that the update phase is broken in two subphases: the first, where the “boundary,” which is involved in communication, is updated, and the second, where the “core,” which neither use nor provide communicated data, is updated.


while(!converged(A)){
    update_boundary(A);
    MPI_Win_fence(MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
    for(i=0; i < fromneighbors; i++)
        MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
                     todisp[i], 1, totype[i], win);
    MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
}

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 11.7 Same generic example, with more computation/communication overlap. We assume that the update phase is broken in two subphases: the first, where the “boundary,” which is involved in communication, is updated, and the second, where the “core,” which neither use nor provide communicated data, is updated.
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 11.8** Same code as in Example 11.6, rewritten using post-start-complete-wait.

```c
... 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 11.9** Same example, with split phases, as in Example 11.7.

```c
... while(!converged(A)){
  update_boundary(A);
  MPI_Win_post(togroup, MPI_MODE_NOPUT, win);
  MPI_Win_start(fromgroup, 0, win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf[i], 1, totype[i], fromneighbor[i],
            fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
}
```

**Example 11.10** 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 win consists of array A1.

```c
... 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 */
while(!converged(A0, A1)){
    /* communication on A0 and computation on A1 */
    update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
    for(i=0; i < neighbors; i++)
        MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
                fromdisp0[i], 1, fromtype0[i], win0);
    update1(A1); /* local update of A1 that depends on A1 (and A1) */
    MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
    for(i=0; i < neighbors; i++)
        MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
    MPI_Win_complete(win0);MPI_Win_wait(win0);
    /* communication on A1 and computation on A0 */
    update2(A0, A1); /* local update of A1 that depends on A0 (and A1) */
    for(i=0; i < neighbors; i++)
        MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                fromdisp1[i], 1, fromtype1[i], win1);
    update1(A0); /* local update of A0 only, concurrent with communication that updates A1 */
    if (!converged(A0, A1))
        MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
    MPI_Win_complete(win1);MPI_Win_wait(win1);
}

/* communication on A0 and computation on A1 */
while(!converged(A0, A1)){
    /* the barrier is needed because the start call inside the loop uses the nocheck option */
    update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
    for(i=0; i < neighbors; i++)
        MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
                fromdisp0[i], 1, fromtype0[i], win0);
    update1(A1); /* local update of A1 that depends on A0 (and A1) */
    MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
    for(i=0; i < neighbors; i++)
        MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
    MPI_Win_complete(win0);MPI_Win_wait(win0);
    /* communication on A0 and computation on A1 */
    update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
    for(i=0; i < neighbors; i++)
        MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                fromdisp1[i], 1, fromtype1[i], win1);
    update1(A0); /* local update of A0 only, concurrent with communication that updates A1 */
    if (!converged(A0, A1))
        MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
    MPI_Win_complete(win1);MPI_Win_wait(win1);
    /* communication on A1 and computation on A0 */
    update2(A0, A1); /* local update of A0 that depends on A1 only, concurrent with communication that updates A1 */
    if (!converged(A0, A1))
        MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
    MPI_Win_complete(win0);MPI_Win_wait(win0);
    /* communication on A1 and computation on A0 */
    update2(A0, A1); /* local update of A1 that depends on A0 (and A0) */
    for(i=0; i < neighbors; i++)
        MPI_Get(&tobuf1[i], 1, totype1[i], neighbor[i],
                fromdisp1[i], 1, fromtype1[i], win1);
    update1(A0); /* local update of A0 only, concurrent with communication that updates A1 */
    if (!converged(A0, A1))
        MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
    MPI_Win_complete(win1);MPI_Win_wait(win1);
}

A process posts the local window associated with win0 before it completes RMA accesses to the remote windows associated with win1. When the wait(win0) call returns, then all neighbors of the calling process have posted the windows associated with win0. Conversely, when the wait(win0) call returns, then all neighbors of the calling process have posted the windows associated with win1. Therefore, the nocheck option can be used with the calls to MPI_WIN_START.

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 communication. On some systems, a put call may be more efficient than a get call, as it requires information exchange only in one direction.

### 11.6 Error Handling

#### 11.6.1 Error Handlers

Errors occurring during calls to 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 default error handler associated with win is MPI_ERRORS_ARE_FATAL. Users may change this default by explicitly associating a new error handler with win (see Section 8.3, page 264).

11.6.2 Error Classes

The following error classes for one-sided communication are defined:

<table>
<thead>
<tr>
<th>Error Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERR_WIN</td>
<td>invalid win argument</td>
</tr>
<tr>
<td>MPI_ERR_BASE</td>
<td>invalid base argument</td>
</tr>
<tr>
<td>MPI_ERR_SIZE</td>
<td>invalid size argument</td>
</tr>
<tr>
<td>MPI_ERR_DISP</td>
<td>invalid disp argument</td>
</tr>
<tr>
<td>MPI_ERR_LOCKTYPE</td>
<td>invalid locktype argument</td>
</tr>
<tr>
<td>MPI_ERR_ASSERT</td>
<td>invalid assert argument</td>
</tr>
<tr>
<td>MPI_ERR_RMA_CONFLICT</td>
<td>conflicting accesses to window</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SYNC</td>
<td>wrong synchronization of RMA calls</td>
</tr>
</tbody>
</table>

Table 11.1: Error classes in one-sided communication routines

11.7 Semantics and Correctness

The semantics of RMA operations is best understood by assuming that the system maintains a separate public copy of each window, in addition to the original location in process memory (the private window copy). There is only one instance of each variable in process memory, but a distinct public copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 11.5.

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.

1. An RMA operation is completed at the origin by the ensuing call to `MPI_WIN_COMPLETE`, `MPI_WIN_FENCE` or `MPI_WIN_UNLOCK` 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.

The following error classes for one-sided communication are defined:

<table>
<thead>
<tr>
<th>Error Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERR_WIN</td>
<td>invalid win argument</td>
</tr>
<tr>
<td>MPI_ERR_BASE</td>
<td>invalid base argument</td>
</tr>
<tr>
<td>MPI_ERR_SIZE</td>
<td>invalid size argument</td>
</tr>
<tr>
<td>MPI_ERR_DISP</td>
<td>invalid disp argument</td>
</tr>
<tr>
<td>MPI_ERR_LOCKTYPE</td>
<td>invalid locktype argument</td>
</tr>
<tr>
<td>MPI_ERR_ASSERT</td>
<td>invalid assert argument</td>
</tr>
<tr>
<td>MPI_ERR_RMA_CONFLICT</td>
<td>conflicting accesses to window</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SYNC</td>
<td>wrong synchronization of RMA calls</td>
</tr>
</tbody>
</table>

Table 11.1: Error classes in one-sided communication routines

11.7 Semantics and Correctness

The semantics of RMA operations is best understood by assuming that the system maintains a separate public copy of each window, in addition to the original location in process memory (the private window copy). There is only one instance of each variable in process memory, but a distinct public copy of the variable for each window that contains it. A load accesses the instance in process memory (this includes MPI sends). A store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 11.5.

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.

1. An RMA operation is completed at the origin by the ensuing call to `MPI_WIN_COMPLETE`, `MPI_WIN_FENCE` or `MPI_WIN_UNLOCK` 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 then the operation is completed at the target by that same call to MPI_WIN_UNLOCK.

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, or MPI_WIN_UNLOCK is executed on that window by the window owner.

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, or MPI_WIN_LOCK is executed on that window by the window owner.

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 executes MPI_WIN_UNLOCK. On the other hand, the update of private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed until the process executes a suitable synchronization call. Updates to a public window copy can also be delayed until the window owner executes a synchronization call, if fences or post-start-complete-wait synchronization is used. Only when lock synchronization is used does it becomes necessary to update the public window copy, even if the window owner does not execute any related synchronization call.

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 then the operation is completed at the target by that same call to MPI_WIN_UNLOCK.

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, or MPI_WIN_UNLOCK is executed on that window by the window owner.

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, or MPI_WIN_LOCK is executed on that window by the window owner.

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 executes MPI_WIN_UNLOCK. On the other hand, the update of private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed until the process executes a suitable synchronization call. Updates to a public window copy can also be delayed until the window owner executes a synchronization call, if fences or post-start-complete-wait synchronization is used. Only when lock synchronization is used does it becomes necessary to update the public window copy, even if the window owner does not execute any related synchronization 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 \texttt{MPI\_WIN\_FENCE(0, win1)} by the window owner makes visible in the process memory previous updates to window \texttt{win\_1} by remote processes. A subsequent call to \texttt{MPI\_WIN\_FENCE(0, win2)} makes these updates visible in the public copy of \texttt{win2}.

A correct program must obey the following rules.

1. A location in a window must not be accessed locally once an update to that location has started, until the update becomes visible in the private window copy in process memory.

2. 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 that use the same operation, with the same predefined datatype, on the same window.

3. A put or accumulate must not access a target window once a local update or a put or accumulate update to another (overlapping) target window have started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a local update in process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

A program is erroneous if it violates these rules.

\textit{Rationale.} The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were locally updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library will have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (End of rationale.)

\textit{Advice to users.} A user can write correct programs by following the following rules:

\textbf{fence:} During each period between fence calls, each window is either updated by put or accumulate calls, or updated by local 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.

\textbf{post-start-complete-wait:} A window should not be updated locally while being 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.

A user can write correct programs by following the following rules:

\textbf{fence:} During each period between fence calls, each window is either updated by put or accumulate calls, or updated by local 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.

\textbf{post-start-complete-wait:} A window should not be updated locally while being 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.

A correct program must obey the following rules.

1. A location in a window must not be accessed locally once an update to that location has started, until the update becomes visible in the private window copy in process memory.

2. 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 that use the same operation, with the same predefined datatype, on the same window.

3. A put or accumulate must not access a target window once a local update or a put or accumulate update to another (overlapping) target window have started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a local update in process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window.

A program is erroneous if it violates these rules.

\textit{Rationale.} The last constraint on correct RMA accesses may seem unduly restrictive, as it forbids concurrent accesses to nonoverlapping locations in a window. The reason for this constraint is that, on some architectures, explicit coherence restoring operations may be needed at synchronization points. A different operation may be needed for locations that were locally updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library will have to track precisely which locations in a window were updated by a put or accumulate call. The additional overhead of maintaining such information is considered prohibitive. (End of rationale.)

\textit{Advice to users.} A user can write correct programs by following the following rules:

\textbf{fence:} During each period between fence calls, each window is either updated by put or accumulate calls, or updated by local 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.

\textbf{post-start-complete-wait:} A window should not be updated locally while being 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 local 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-complete-wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK 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. (End of advice to users.)

### 11.7.1 Atomicity

The outcome of concurrent accumulates to the same location, with the same operation and predefined datatype, is as if the accumulates where done at that location in some serial order. On the other hand, if two locations are both updated by two accumulate calls, then the updates may occur in reverse order at the two locations. Thus, there is no guarantee that the entire call to MPI_ACCUMULATE is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to MPI_ACCUMULATE, cannot be accessed by load or an RMA call other than accumulate, until the MPI_ACCUMULATE call has completed (at the target). Different interleavings can lead to different results only to the extent that computer arithmetics are not truly associative or commutative.

### 11.7.2 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled, then 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

(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 local 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-complete-wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK 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. (End of advice to users.)
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 11.4, on page 339. 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 occur, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 11.5, on page 344. 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 11.6. 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 these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

Assume, in the last example, that the order of the post and start calls is reversed, at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur. Similarly, the program will deadlock, if the order of the complete and wait calls is reversed, at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice-versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait 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 11.4, on page 339. 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 occur, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 11.5, on page 344. 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 11.6. 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 these are done, the wait calls should complete at both processes. Thus, this communication should not deadlock, irrespective of the amount of data transferred.

Assume, in the last example, that the order of the post and start calls is reversed, at each process. Then, the code may deadlock, as each process may block on the start call, waiting for the matching post to occur. Similarly, the program will deadlock, if the order of the complete and wait calls is reversed, at each process.

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice-versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait
of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 11.8. This 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 11.8, 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.

A similar issue is whether such progress must occur while a process is busy computing, or blocked in a non-MPI call. Suppose that in the last example the send-receive 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 replaced by a very long compute loop. Then, according to one interpretation of the MPI standard, process 0 must return from the complete call after a bounded delay, even if process 1 does not reach any MPI call in this period of time. According to another interpretation, the complete call may block until process 1 reaches the wait call, or reaches another MPI call. The qualitative behavior is the same, under both interpretations, unless a process is caught in an infinite compute loop, in which case the difference may not matter. However, the quantitative expectations are different. Different MPI implementations reflect these different interpretations. While this ambiguity is unfortunate, it does not seem to affect many real codes.

11.7.3 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 value of these variables. An RMA call may access a variable in memory (or cache), while the
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.

The problem is illustrated by the following code:

<table>
<thead>
<tr>
<th>Source of Process 1</th>
<th>Source of Process 2</th>
<th>Executed in Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bbbb = 777</code></td>
<td><code>buff = 999</code></td>
<td><code>reg_A:=999</code></td>
</tr>
<tr>
<td>call MPI_WIN_FENCE</td>
<td>call MPI_WIN_FENCE</td>
<td><code>stop appl. thread</code></td>
</tr>
<tr>
<td>`call MPI_PUT(bbbb)</td>
<td><code>call MPI_WIN_FENCE</code></td>
<td><code>buff:=777 in PUT handler</code></td>
</tr>
<tr>
<td>into buff of process 2)</td>
<td><code>continue appl. thread</code></td>
<td></td>
</tr>
<tr>
<td><code>call MPI_WIN_FENCE</code></td>
<td><code>call MPI_WIN_FENCE</code></td>
<td><code>ccc = buff</code></td>
</tr>
<tr>
<td><code>ccc := reg_A</code></td>
<td></td>
<td><code>ccc := reg_A</code></td>
</tr>
</tbody>
</table>

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`.

This problem, which also afflicts in some cases send/receive communication, is discussed more at length in Section 16.2.2.

MPI implementations will avoid this problem for standard conforming C programs. 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 `COMMON` blocks, or to variables that were declared `VOLATILE` (while `VOLATILE` is not a standard Fortran declaration, it is supported by many Fortran compilers). Details and an additional solution are discussed in Section 16.2.2, “A Problem with Register Optimization,” on page 466. See also, “Problems Due to Data Copying and Sequence Association,” on page 463, for additional Fortran problems.
Chapter 12

External Interfaces

12.1 Introduction

This chapter begins with calls used to create generalized requests, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. This can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in status. This is needed for generalized requests.

The chapter continues, in Section 12.4, with a discussion of how threads are to be handled in MPI. Although thread compliance is not required, the standard specifies how threads are to work if they are provided.

12.2 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 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_CANCEr 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 very 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. (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.

Chapter 12

External Interfaces

12.1 Introduction

This chapter begins with calls used to create generalized requests, which allow users to create new nonblocking operations with an interface similar to what is present in MPI. This can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in status. This is needed for generalized requests.

The chapter continues, in Section 12.4, with a discussion of how threads are to be handled in MPI. Although thread compliance is not required, the standard specifies how threads are to work if they are provided.

12.2 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 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_CANCEr 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 very 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. (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 when the operation completes. This
is done by making a call to `MPI_GREQUEST_COMPLETE`. MPI maintains the “completion”
status of generalized requests. Any other request state has to be maintained by the user.

A new generalized request is started with

```c
MPI_GREQUEST_START(query_fn, free_fn, cancel_fn, extra_state, request)
```

### Parameters

- `query_fn`: callback function invoked when request status is queried (function)
- `free_fn`: callback function invoked when request is freed (function)
- `cancel_fn`: callback function invoked when request is cancelled (function)
- `extra_state`: extra state
- `request`: generalized request (handle)

### Return Value

The call results in a handle to the generalized request. The syntax and meaning of the callback functions are listed below. All callback functions are passed the `extra_state` argument that was associated with the request by the starting call `MPI_GREQUEST_START`. This can be used to maintain user-defined state for the request.

In C, the query function is

```c
typedef int MPI_Grequest_query_function(void *extra_state,
                                      MPI_Status *status);
```

### Advice to users

Note that a generalized request belongs, in C++, to the class `MPI::Grequest`, which is a derived class of `MPI::Request`. It is of the same type as regular requests, in C and Fortran. (End of advice to users.)

The call starts a generalized request and returns a handle to it in `request`. The syntax and meaning of the callback functions are listed below. All callback functions are passed the `extra_state` argument that was associated with the request by the starting call `MPI_GREQUEST_START`. This can be used to maintain user-defined state for the request.

In C, the query function is

```c
typedef int MPI_Grequest_query_function(void *extra_state,
                                      MPI_Status *status);
```
12.2. GENERALIZED REQUESTS

in Fortran

SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

and in C++

typedef int MPI::Grequest::Query_function(void* extra_state, 
  MPI::Status& status);

query_fn function computes the status that should be returned for the 
generalized request. The status also includes information about successful/unsuccessful cancellation of the request (result to be returned by MPI_TEST_CANCELLED).

query_fn callback is invoked by the MPI_WAIT{TEST}{ANY|SOME|ALL} call that 
completed the generalized request associated with this callback. The callback function is also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when 
the call occurs. In both cases, the callback is passed a reference to the corresponding status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or 
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI 
will pass a valid status object to query_fn, and this status will be ignored upon return of the 
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE is 
called on the request; it may be invoked several times for the same generalized request, 
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also that a call to MPI_[WAIT][TEST]{SOME|ALL} may cause multiple invocations of query_fn 
callback functions, one for each generalized request that is completed by the MPI call. The 
order of these invocations is not specified by MPI.

In C, the free function is

typedef int MPI_Grequest_free_function(void *extra_state);

and in Fortran

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
  INTEGER IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

and in C++

typedef int MPI::Grequest::Free_function(void* extra_state, 
  MPI::Status& status);

free_fn function is invoked to clean up user-allocated resources when the generalized 
request is freed.

free_fn callback is invoked by the MPI_[WAIT][TEST]{ANY|SOME|ALL} call that 
completed the generalized request associated with this callback. free_fn is invoked after the call 
to query_fn for the same request. However, if the MPI call completed multiple generalized 
requests, the order in which free_fn callbacks are invoked is not specified by MPI.

free_fn callback is also invoked for generalized requests that are freed by a call to 
MPI_REQUEST_FREE (no call to WAIT_[|TEST]SOME|ALL) will occur for such a request). In this case, the callback function will be called either in the MPI call 
MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), 
whichever happens last, i.e., in this case the actual freeing code is executed as soon as both

in Fortran

SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

and in C++

typedef int MPI::Grequest::Query_function(void* extra_state, 
  MPI::Status& status);

query_fn function computes the status that should be returned for the 
generalized request. The status also includes information about successful/unsuccessful cancellation of the request (result to be returned by MPI_TEST_CANCELLED).

query_fn callback is invoked by the MPI_WAIT{TEST}{ANY|SOME|ALL} call that 
completed the generalized request associated with this callback. The callback function is also invoked by calls to MPI_REQUEST_GET_STATUS, if the request is complete when 
the call occurs. In both cases, the callback is passed a reference to the corresponding status variable passed by the user to the MPI call; the status set by the callback function is returned by the MPI call. If the user provided MPI_STATUS_IGNORE or 
MPI_STATUSES_IGNORE to the MPI function that causes query_fn to be called, then MPI 
will pass a valid status object to query_fn, and this status will be ignored upon return of the 
callback function. Note that query_fn is invoked only after MPI_GREQUEST_COMPLETE is 
called on the request; it may be invoked several times for the same generalized request, 
e.g., if the user calls MPI_REQUEST_GET_STATUS several times for this request. Note also that a call to MPI_[WAIT][TEST]{SOME|ALL} may cause multiple invocations of query_fn 
callback functions, one for each generalized request that is completed by the MPI call. The 
order of these invocations is not specified by MPI.

In C, the free function is

typedef int MPI_Grequest_free_function(void *extra_state);

and in Fortran

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
  INTEGER IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

and in C++

typedef int MPI::Grequest::Free_function(void* extra_state, 
  MPI::Status& status);

free_fn function is invoked to clean up user-allocated resources when the generalized 
request is freed.

free_fn callback is invoked by the MPI_[WAIT][TEST]{ANY|SOME|ALL} call that 
completed the generalized request associated with this callback. free_fn is invoked after the call 
to query_fn for the same request. However, if the MPI call completed multiple generalized 
requests, the order in which free_fn callbacks are invoked is not specified by MPI.

free_fn callback is also invoked for generalized requests that are freed by a call to 
MPI_REQUEST_FREE (no call to WAIT_[|TEST]SOME|ALL) will occur for such a request). In this case, the callback function will be called either in the MPI call 
MPI_REQUEST_FREE(request), or in the MPI call MPI_GREQUEST_COMPLETE(request), 
whichever happens last, i.e., in this case the actual freeing code is executed as soon as both
calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request
is not deallocated until after free_fn completes. Note that free_fn will be invoked only once
per request by a correct program.

Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle
that is not deallocated until after free_fn completes. Users should note that free will deallocate the object after free_fn completes. At this point, user copies of the request handle no longer point to a valid request. MPI
will not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to
avoid accessing this stale handle. This is a special case where MPI defers deallocating
the object until a later time that is known by the user. (End of advice to users.)

In C, the cancel function is
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
in Fortran
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
   LOGICAL COMPLETE
and in C++
typedef int MPI::Grequest::Cancel_function(void* extra_state,
        bool complete);

cancel_fn function is invoked to start the cancelation of a generalized request. It is
called by MPI_CANCEL(request). MPI passes to the callback function complete=true 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
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_STATES_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
calls MPI_REQUEST_FREE and MPI_GREQUEST_COMPLETE have occurred. The request
is not deallocated until after free_fn completes. Note that free_fn will be invoked only once
per request by a correct program.

Advice to users. Calling MPI_REQUEST_FREE(request) will cause the request handle
that is not deallocated until after free_fn completes. Users should note that free will deallocate the object after free_fn completes. At this point, user copies of the request handle no longer point to a valid request. MPI
will not set user copies to MPI_REQUEST_NULL in this case, so it is up to the user to
avoid accessing this stale handle. This is a special case where MPI defers deallocating
the object until a later time that is known by the user. (End of advice to users.)

In C, the cancel function is
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
in Fortran
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
   LOGICAL COMPLETE
and in C++
typedef int MPI::Grequest::Cancel_function(void* extra_state,
        bool complete);

cancel_fn function is invoked to start the cancelation of a generalized request. It is
called by MPI_CANCEL(request). MPI passes to the callback function complete=true 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
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_STATES_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 in the error field of status the returned error code. (End of advice to users.)

MPI_GREQUEST_COMPLETE(request)

int MPI_Grequest_complete(MPI_Request request)

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, all 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.)

12.2. GENERALIZED REQUESTS

Example 12.1 This example shows the code for a user-defined reduce operation on an int using a binary tree: each non-root node receives two messages, sums them, and sends them up. We assume that no status is returned and that the operation cannot be cancelled.

typedef struct {
    MPI_Comm comm;
    int tag;
    int root;
    int valin;
    int *valout;
    MPI_Request request;
} ARGS;
int myreduce(MPI_Comm comm, int tag, int root,
    int valin, int *valout, MPI_Request *request)
{
    ARGS *args;
    pthread_t thread;
    /* start request */
    MPI_Grequest_start(query_fn, free_fn, cancel_fn, NULL, request);

    args = (ARGS*)malloc(sizeof(ARGS));
    args->comm = comm;
    args->tag = tag;
    args->root = root;
    args->valin = valin;
    args->valout = valout;
    args->request = *request;
    /* spawn thread to handle request */
    pthread_create(&thread, NULL, reduce_thread, args);
    return MPI_SUCCESS;
}

/* thread code */
void* reduce_thread(void *ptr)
{
    int lchild, rchild, parent, lval, rval, val;
    MPI_Request req[2];
    ARGS *args;
    args = (ARGS*)ptr;
    /* compute left, right child and parent in tree; set
    to MPI_PROC_NULL if does not exist */
    /* code not shown */
    ...
    MPI_Irecv(&lval, 1, MPI_INT, lchild, args->tag, args->comm, &req[0]);
    MPI_Irecv(&rval, 1, MPI_INT, rchild, args->tag, args->comm, &req[1]);
    MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
    val = lval + args->valin + rval;
    MPI_Send(&val, 1, MPI_INT, parent, args->tag, args->comm);
    if (parent == MPI_PROC_NULL) *(args->valout) = val;
    MPI_Grequest_complete((args->request));
    free(ptr);
    return(NULL);
}
int query_fn(void *extra_state, MPI_Status *status) {
    /* always send just one int */
    MPI_Status_set_elements(status, MPI_INT, 1);
    /* can never cancel so always true */
    status->MPI_TAG = MPI_UNDEFINED;
    /* this generalized request never fails */
    return MPI_SUCCESS;
}

int free_fn(void *extra_state) {
    /* this generalized request does not need to do any freeing */
    /* as a result it never fails here */
    return MPI_SUCCESS;
}

int cancel_fn(void *extra_state, int complete) {
    /* This generalized request does not support cancelling.
     * Abort if not already done. If done then treat as if cancel failed.*/
    if (!complete) {
        fprintf(stderr, "Cannot cancel generalized request - aborting program\n");
        MPI_Abort(MPI_COMM_WORLD, 99);
    }
    return MPI_SUCCESS;
}

12.3 Associating Information with Status

MPI supports several different types of requests besides those for point-to-point operations. These range from MPI calls for I/O to generalized requests. It is desirable to allow these calls use the same request mechanism. This allows one to wait or test on different types of requests. However, MPI_{TEST|WAIT}_{ANY|SOME|ALL} returns a status with information about the request. With the generalization of requests, one needs to define what information will be returned in the status object. Each 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 value 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 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:

```c
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype, int count);
```

This call modifies the opaque part of status so that a call to `MPI_GET_ELEMENTS` will return count. `MPI_GET_COUNT` will return a compatible value.

**Rationale.** The number of elements is set instead of the count because the former can deal with a nonintegral number of datatypes. (End of rationale.)

A subsequent call to `MPI_GET_COUNT` or to `MPI_GET_ELEMENTS` 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`.

**Rationale.** This is similar to the restriction that holds when count is set by a receive operation: in that case, the calls to `MPI_GET_COUNT` and `MPI_GET_ELEMENTS` must use a datatype with the same signature as the datatype used in the receive call. (End of rationale.)

```c
int MPI_Status_set_cancelled(MPI_Status *status, int flag);
```

**Rationale.** This is similar to the restriction that holds when count is set by a receive operation: in that case, the calls to `MPI_GET_COUNT` and `MPI_GET_ELEMENTS` must use a datatype with the same signature as the datatype used in the receive call. (End of rationale.)
12.4. MPI and Threads

This section specifies the interaction between MPI calls and threads. 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, it is not required that all MPI implementations fulfill all the requirements specified in this section.

This section generally assumes a thread package similar to POSIX threads [29], but the syntax and semantics of thread calls are not specified here — these are beyond the scope of this document.

12.4.1 General

In a thread-compliant implementation, an MPI process is a process that may be multi-threaded. 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 multi-threaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations where MPI ‘processes’ are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their ‘processes’ are single-threaded).

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.

Rationale. This model corresponds to the POSIX model of interprocess communication: the fact that a process is multi-threaded, rather than single-threaded, does not affect the external interface of this process. MPI implementations where MPI ‘processes’ are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their ‘processes’ are single-threaded).

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 12.2 Process 0 consists of two threads. The first thread executes a blocking send call MPI_Send(buf1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI_Recv(buf2, 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.)

12.4.2 Clarifications

Initialization and Completion. The call to MPI_FINALIZE should occur on the same thread that initialized MPI. We call this thread the main thread. The call should occur only after all the process threads have completed their MPI calls, and have no pending communications or I/O operations.

Rationale. This constraint simplifies implementation. (End of rationale.)

Multiple threads completing the same request. A program where two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of calls. In MPI, a request can only be completed once. Any combination of wait or test which violates this rule is erroneous.

Rationale. This 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.

**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 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 filehandle 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 already 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.** 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.)*

**Exception handlers.** An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call; the exception handler may be executed by a thread that 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 exception handler to be executed on the thread where the exception occurred. *(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.

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.

**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 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 filehandle 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 already 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.** 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.)*

**Exception handlers.** An exception handler does not necessarily execute in the context of the thread that made the exception-raising MPI call; the exception handler may be executed by a thread that 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 exception handler to be executed on the thread where the exception occurred. *(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 where 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.)

12.4.3 Initialization

The following function may be used to initialize MPI, and initialize the MPI thread environment, instead of `MPI_INIT`.

```c
int MPI_Init_thread(int *argc, char **argv, int required, int *provided)
```

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 multi-threaded, 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 370).
```

Advice to users. In C and C++, the passing of `argc` and `argv` is optional. In C, this is accomplished by passing the appropriate null pointer. In C++, this is accomplished with two separate bindings to cover these two cases. This is as with `MPI_INIT` as discussed in Section 8.7. (End of advice to users.)

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 multi-threaded, 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 370).
```
MPI_THREAD_SERIALIZED The process may be multi-threaded, 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.

These values are monotonic: i.e., MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE.

Different processes in MPI_COMM_WORLD may require different levels of thread support.
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.

A thread compliant MPI implementation will be able to return provided = MPI_THREAD_MULTIPLE. Such an implementation may always return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required. At the other extreme, an MPI library that is not thread compliant may always return provided = MPI_THREAD_SINGLE, irrespective of the value of required.

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 thread support available when the MPI program is started, e.g., with arguments to mpiexec. This will affect the outcome of calls to MPI_INIT and MPI_INIT_THREAD. Suppose, for example, that an MPI program has been started so that only MPI_THREAD_MULTIPLE is available. Then MPI_INIT_THREAD will return provided = MPI_THREAD_MULTIPLE, irrespective of the value of required; a call to MPI_INIT will also initialize the MPI thread support level to MPI_THREAD_MULTIPLE. Suppose, on the other hand, that an MPI program has been started so that all four levels of thread support are available. Then, a call to MPI_INIT_THREAD will return provided = required; on the other hand, a call to MPI_INIT will initialize the MPI thread support level to MPI_THREAD_SINGLE.

Rationale. Various optimizations are possible when MPI code is executed single-threaded, or is executed on multiple threads, but not concurrently: mutual exclusion code may be omitted. Furthermore, if only one thread executes, then the MPI library can use library functions that are not thread safe, without risking conflicts with user threads. Also, the model of one communication thread, multiple computation threads fits many applications well, e.g., if the process code is a sequential Fortran/C/C++ program with MPI calls that has been parallelized by a compiler for execution on an SMP node, in a cluster of SMPs, then the process computation is multi-threaded, but MPI calls will likely execute on a single thread.

The design accommodates a static specification of the thread support level, for environments that require static binding of libraries, and for compatibility for current multi-threaded MPI codes. (End of rationale.)
Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C/ C++/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. (End of advice to implementors.)

The following function can be used to query the current level of thread support.

MPI_QUERY_THREAD(provided)

OUT provided level of thread support (integer)

int MPI_Query_thread(int *provided)
MPI_QUERY_THREAD(PROVIDED, IERROR)

int MPI::Query_thread()

The call returns in provided the current level of thread support. This will be the value returned in provided by MPI_INIT_THREAD if MPI was initialized by a call to MPI_INIT_THREAD().

MPI_IS_THREAD_MAIN(flag)

OUT flag true if calling thread is main thread, false otherwise (logical)

int MPI_Is_thread_main(int *flag)
MPI_IS_THREAD_MAIN(FLAG, IERROR)

bool MPI::Is_thread_main()

This function can be called by a thread to find out whether it is the main thread (the thread that called MPI_INIT or MPI_INIT_THREAD). All routines listed in this section must be supported by all MPI implementations.

Rationale. MPI libraries are required to provide these calls even if they do not support threads, so that portable code that contains invocations to these functions be able to link correctly. MPI_INIT continues to be supported so as to provide compatibility with current MPI codes. (End of rationale.)

Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI call other than MPI_INITIALIZED should be executed by these threads, until

Advice to implementors. If provided is not MPI_THREAD_SINGLE then the MPI library should not invoke C/ C++/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. (End of advice to implementors.)

The following function can be used to query the current level of thread support.

MPI_QUERY_THREAD(provided)

OUT provided level of thread support (integer)

int MPI_Query_thread(int *provided)
MPI_QUERY_THREAD(PROVIDED, IERROR)

int MPI::Query_thread()

The call returns in provided the current level of thread support. This will be the value returned in provided by MPI_INIT_THREAD if MPI was initialized by a call to MPI_INIT_THREAD().

MPI_IS_THREAD_MAIN(flag)

OUT flag true if calling thread is main thread, false otherwise (logical)

int MPI_Is_thread_main(int *flag)
MPI_IS_THREAD_MAIN(FLAG, IERROR)

bool MPI::Is_thread_main()

This function can be called by a thread to find out whether it is the main thread (the thread that called MPI_INIT or MPI_INIT_THREAD). All routines listed in this section must be supported by all MPI implementations.

Rationale. MPI libraries are required to provide these calls even if they do not support threads, so that portable code that contains invocations to these functions be able to link correctly. MPI_INIT continues to be supported so as to provide compatibility with current MPI codes. (End of rationale.)

Advice to users. It is possible to spawn threads before MPI is initialized, but no MPI call other than MPI_INITIALIZED should be executed by these threads, until
MPI_INIT_THREAD is invoked by one thread (which, thereby, becomes the main thread). In particular, it is possible to enter the MPI execution with a multi-threaded process.

The level of thread support provided is a global property of the MPI process that can be specified only once, when MPI is initialized on that process (or before). Portable third party libraries have to be written so as to accommodate any provided level of thread support. Otherwise, their usage will be restricted to specific level(s) of thread support. If such a library can run only with specific level(s) of thread support, e.g., only with MPI_THREAD_MULTIPLE, then MPI_QUERY_THREAD can be used to check whether the user initialized MPI to the correct level of thread support and, if not, raise an exception. (End of advice to users.)
Chapter 13

I/O

13.1 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 [35], collective
buffering [6, 13, 36, 39, 46], and disk-directed I/O [31]) can only be implemented if the par-
allel 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 sup-
port 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.

13.1.1 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 displace-
ments are nonnegative 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.
**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 nonnegative 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 13.1 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).

![Figure 13.1: Etypes and filetypes](image1)

A group of processes can use complementary views to achieve a global data distribution such as a scatter/gather pattern (see Figure 13.2).

![Figure 13.2: Partitioning a file among parallel processes](image2)

**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 13.2 is the position of the 8th etype in the file after the displacement. An “explicit offset” is an offset that is used as a formal parameter in explicit data access routines.

![Figure 13.2: Partitioning a file among parallel processes](image3)
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.

13.2 File Manipulation

13.2.1 Opening a File

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN   comm   communicator (handle)
IN   filename   name of file to open (string)
IN   amode   file access mode (integer)
IN   info   info object (handle)
OUT  fh   new file handle (handle)

int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN   comm   communicator (handle)
IN   filename   name of file to open (string)
IN   amode   file access mode (integer)
IN   info   info object (handle)
OUT  fh   new file handle (handle)

int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)

static MPI::File MPI::File::Open(const MPI::Intracomm& comm, const char* filename, int amode, const MPI::Info& info)

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 the same file. (Values for info may vary.) comm must be an intracommunicator; it is erroneous to pass an intercommunicator to MPI_FILE_OPEN. Errors in MPI_FILE_OPEN are raised using the default file error handler (see Section 13.7, page 429). A process can open a file independently of other processes by using the MPI_COMM_SELF communicator. The file handle returned, fh, can be subsequently used to access the file until the file is closed using MPI_FILE_CLOSE. Before calling MPI_FINALIZE, the user is required to close (via MPI_FILE_CLOSE) all files that were opened with MPI_FILE_OPEN. Note that the communicator comm is unaffected by MPI_FILE_OPEN and continues to be usable in all processes. 

13.2.2 Closing a File

MPI_FILE_CLOSE(fh)

OUT fh   file handle

int MPI_File_close(MPI_File fh)

13.2.3 Reading and Writing

MPI_READ(fh, type, disp, len, out)

IN fh   file handle
IN type   etype
IN disp   displacement
IN len   length
OUT out   output buffer

MPI_WRITE(fh, type, disp, len, in)

IN fh   file handle
IN type   etype
IN disp   displacement
IN len   length
OUT in   input buffer

13.2.4 Other Operations on a File

MPI_QUERY(fh, info)

IN fh   file handle
OUT info   query info

13.2.5 Internals

13.2.5.1 Accessing Internals

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.
MPI routines (e.g., MPI_SEND). Furthermore, the use of comm will not interfere with I/O behavior.

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.)

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/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 (non-portably) 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.)

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.)

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_CREAT — 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/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 (non-portably) 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.)
13.2. FILE MANIPULATION

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 [29]. 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 13.2.8, page 382). 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 13.6.1, page 420). The more stringent atomic mode consistency semantics, required for atomicity of conflicting accesses, can be set using MPI_FILE_SET_ATOMICITY.

13.2.2 Closing a File

MPI_FILE_CLOSE(fh)
INOUT fh file handle (handle)

int MPI_File_close(MPI_File *fh)
MPI_FILE_CLOSE(fh, IERROR)
INTEGER fh, IERROR
**Chapter 13. I/O**

```c
void MPI::File::Close()

MPI_FILE_CLOSE first synchronizes file state (equivalent to performing an
MPI_FILE_SYNC), then closes the file associated with fh. The file is deleted if it was
opened with access mode MPI_MODE_DELETE_ON_CLOSE (equivalent to performing an
MPI_FILE_DELETE). MPI_FILE_CLOSE is a collective routine.

Advice to users. If the file is deleted on close, and there are other processes currently
accessing the file, the status of the file and the behavior of future accesses by these
processes are implementation dependent. (End of advice to users.)

The user is responsible for ensuring that all outstanding nonblocking requests and
split collective operations associated with fh made by a process have completed before that
process calls MPI_FILE_CLOSE.

The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
MPI_FILE_NULL.

### 13.2.3 Deleting a File

```c
MPI_FILE_DELETE(filename, info)
```

- **IN** filename name of file to delete (string)
- **IN** info info object (handle)

```c
int MPI_File_delete(char *filename, MPI_Info info)
```

- **CHARACTER**(*) `FILENAME`
- **INTEGER** `INFO`, `IERROR`

```c
static void MPI::File::Delete(const char* filename, const MPI::Info& info)
```

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 13.2.8, page 382). 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 error handler (see Section 13.7, page 429).

The user is responsible for ensuring that all outstanding nonblocking requests and
split collective operations associated with fh made by a process have completed before that
process calls MPI_FILE_CLOSE.

The MPI_FILE_CLOSE routine deallocates the file handle object and sets fh to
MPI_FILE_NULL.

### 13.2.3 Deleting a File

```c
MPI_FILE_DELETE(filename, info)
```

- **IN** filename name of file to delete (string)
- **IN** info info object (handle)

```c
int MPI_File_delete(char *filename, MPI_Info info)
```

- **CHARACTER**(*) `FILENAME`
- **INTEGER** `INFO`, `IERROR`

```c
static void MPI::File::Delete(const char* filename, const MPI::Info& info)
```

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 13.2.8, page 382). 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 error handler (see Section 13.7, page 429).
13.2.4 Resizing a File

MPI_FILE_SET_SIZE(fh, size)

void MPI::File::Set_size(MPI::Offset size)

MPI_FILE_SET_SIZE(fh, size)

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 legal, and equivalent to seeking beyond the current end of file. (End of advice to users.)

All nonblocking requests and split collective operations on fh must be completed before calling MPI_FILE_SET_SIZE. Otherwise, calling MPI_FILE_SET_SIZE is erroneous. As far as consistency semantics are concerned, MPI_FILE_SET_SIZE is a write operation that conflicts with operations that access bytes at displacements between the old and new file sizes (see Section 13.6.1, page 420).

13.2.5 Preallocating Space for a File

MPI_FILE_PREALLOCATE(fh, size)

int MPI_File_preallocate(MPI_File fh, MPI_Offset size)

13.2.5 Preallocating Space for a File

MPI_FILE_PREALLOCATE(fh, size)

int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
void MPI::File::Preallocate(MPI::Offset size)

MPI_FILE_PREALLOCATE ensures that storage space is allocated for the first size bytes of the file associated with fh. MPI_FILE_PREALLOCATE is collective; all processes in the group must pass identical values for size. Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, MPI_FILE_PREALLOCATE has the same effect as writing undefined data. If size is larger than the current file size, the file size increases to size. If size is less than or equal to the current file size, the file size is unchanged.

The treatment of file pointers, pending nonblocking accesses, and file consistency is the same as with MPI_FILE_SET_SIZE. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

Advice to users. In some implementations, file preallocation may be expensive. (End of advice to users.)

13.2.6 Querying the Size of a File

MPI_FILE_GET_SIZE(fh, size)
IN fh file handle (handle)
OUT size size of the file in bytes (integer)

int MPI_File_get_size(MPI_File fh, MPI_Offset *size)

MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a data access operation (see Section 13.6.1, page 420).

13.2.7 Querying File Parameters

MPI_FILE_GET_GROUP(fh, group)
IN fh file handle (handle)
OUT group group which opened the file (handle)

int MPI_File_get_group(MPI_File fh, MPI_Group *group)

MPI_FILE_GET_SIZE(fh, size)
IN fh file handle (handle)
OUT size size of the file in bytes (integer)

int MPI_File_get_size(MPI_File fh, MPI_Offset *size)

MPI_FILE_GET_SIZE returns, in size, the current size in bytes of the file associated with the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_SIZE is a data access operation (see Section 13.6.1, page 420).

13.2.7 Querying File Parameters
13.2. FILE MANIPULATION

MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
INTEGER FH, GROUP, IERROR
MPI::Group MPI::File::Get_group() const

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.

MPI_FILE_GET_AMODE(fh, amode)
IN fh file handle (handle)
OUT amode file access mode used to open the file (integer)

int MPI_File_get_amode(MPI_File fh, int *amode)
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
INTEGER FH, AMODE, IERROR

Example 13.1 In Fortran 77, decoding an amode bit vector will require a routine such as the following:

SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
!! TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE!
!! IF SET, RETURN 1 IN BIT_FOUND, 0 OTHERWISE!
INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
BIT_FOUND = 0
CP_AMODE = AMODE
100 CONTINUE
LBIT = 0
HIFOUND = 0
DO 20 L = MAX_BIT, 0, -1
MATCHER = 2**L
IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
HIFOUND = 1
LBIT = MATCHER
CP_AMODE = CP_AMODE - MATCHER
END IF
20 CONTINUE
IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
IF (BIT_FOUND .EQ. 0 .AND. HIFOUND .EQ. 1 .AND. &
CP_AMODE .GT. 0) GO TO 100
END
This routine could be called successively to decode `amode`, one bit at a time. For example, the following code fragment would check for `MPI_MODE_RDONLY`.

```fortran
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
```

### 13.2.8 File Info

Hints specified via info (see Section 9, page 287) 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. However, hints do not change the semantics of any of the I/O interfaces. In other words, an implementation is free to ignore all hints. 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. 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.)

```fortran
MPI_FILE_SET_INFO(fh, info)
```

**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, an implementation may ignore hints issued in this call that it would have accepted in an open call. (End of advice to users.)

### 13.2.8 File Info

Hints specified via info (see Section 9, page 287) 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. However, hints do not change the semantics of any of the I/O interfaces. In other words, an implementation is free to ignore all hints. 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. 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.)

```fortran
MPI_FILE_SET_INFO(fh, info)
```
nodes

buffer

parameters are further directed via additional hints:

Collective buffering

false

and

ture

into large disk accesses. Legal values for this key are

Collective buffering

the group by a number of target nodes. These target nodes coalesce small requests

on collective accesses. Accesses to the file are performed on behalf of all processes in

benefit from collective buffering. Collective buffering is an optimization performed

This hint specifies whether the application may

Reserved File Hints

Some potentially useful hints (info key values) are outlined below. 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. (For more details on “info,” see Section 9, page 287.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. 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. Legal 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.

Reserved File Hints

Some potentially useful hints (info key values) are outlined below. 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. (For more details on “info,” see Section 9, page 287.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. 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. Legal 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 multidimensional array that is often accessed by subarrays. For this hint to be valid, a hint called cb_block_size (integer) [SAME] must be specified. 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 size 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 that includes MPI_MODE_CREATE. The set of legal values for this key is implementation dependent.

io_node_list (comma separated list of strings) [SAME]: This hint specifies the list of I/O devices that should be used to store the file. This hint is most relevant when the file is created.

nb_proc (integer) [SAME]: This hint specifies the number of parallel processes that will typically be assigned to run programs that access this file. This hint is most relevant when the file is created.

num_io_nodes (integer) [SAME]: This hint specifies the number of I/O devices in the system. This hint is most relevant when the file is created.

striping_factor (integer) [SAME]: This hint specifies the number of I/O devices that the file should be striped across, and is relevant only when the file is created.

striping_unit (integer) [SAME]: This hint specifies the suggested striping unit to be used for this file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.
13.3 File Views

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

INOUT fh file handle (handle)
IN disp displacement (integer)
IN etype elementary datatype (handle)
IN filetype filehandle (filehandle)
IN datarep data representation (string)
IN info info object (handle)

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
MPI_Filetype filetype, char *datarep, MPI_Info info)

MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP

void MPI::File::Set_view(MPI::Offset disp, const MPI::Datatype& etype,
const MPI::Filetype& filetype, const char* datarep,
const MPI::Info& info)

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_FILESET_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, page 11), 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 13.5.1, page 412 for further details.

If MPI_MODE SEQUENTIAL 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.)
CHAPTER 13. I/O

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 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the filetype (see Section 13.5, page 410). (End of advice to users.)

Advice to users. disp can be used to access a different part of the file.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the filetype (see Section 13.5, page 410). (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 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 13.2.8, page 382). The constant MPI_INFO_NULL refers to the null info and can be used when no info needs to be specified.

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 access a different part of the file.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the filetype (see Section 13.5, page 410). (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 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 13.2.8, page 382). 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 13.5, page 410) 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.

MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

| IN  | fh       | file handle (handle) |
| OUT | disp     | displacement (integer) |
| OUT | etype    | elementary datatype (handle) |
| OUT | filetype | filetype (handle) |
| OUT | datarep  | data representation (string) |

int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype, MPI_Datatype *filetype, char *datarep) const

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.

13.4 Data Access

13.4.1 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 (non collective vs. collective). The following combinations of these data access routines, including two types of file pointers (individual and shared) are provided in Table 13.1.
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.

Positioning

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_READ_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 13.4.2, page 390.

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 13.4.3, page 394. 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 13.4.4, page 399.

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

Table 13.1: Data access routines

<table>
<thead>
<tr>
<th>position</th>
<th>sync</th>
<th>coord</th>
<th>noncollective</th>
<th>collective</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit offsets</td>
<td>blocking</td>
<td>MPI_FILE_READ_AT</td>
<td>MPI_FILE_READ_AT_ALL</td>
<td></td>
</tr>
<tr>
<td>nonblocking &amp; split collective</td>
<td>MPI_FILE_READ_AT, MPI_FILE_WRITE_AT, MPI_FILE_WRITE_AT_ALL, MPI_FILE_WRITE_AT_ALL_BEGIN, MPI_FILE_WRITE_AT_ALL_END</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>individual file pointers</td>
<td>blocking</td>
<td>MPI_FILE_READ, MPI_FILE_WRITE, MPI_FILE_WRITE_ALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nonblocking &amp; split collective</td>
<td>MPI_FILE_READ, MPI_FILE_WRITE, MPI_FILE_WRITE_ALL_BEGIN, MPI_FILE_WRITE_ALL_END</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shared file pointer</td>
<td>blocking</td>
<td>MPI_FILE_READ_SHARED, MPI_FILE_WRITE_SHARED, MPI_FILE_WRITE_SHARED_ORDERED, MPI_FILE_WRITE_SHARED_ORDERED_BEGIN, MPI_FILE_WRITE_SHARED_ORDERED_END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

Positioning

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 13.4.2, page 390.

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 13.4.3, page 394. 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 13.4.4, page 399.

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,

\[
new_{\text{file}}_{\text{offset}} = old_{\text{file}}_{\text{offset}} + \frac{\text{elements}(\text{datatype})}{\text{elements}(\text{etype})} \times \text{count}
\]

where \(\text{count}\) is the number of \(\text{datatype}\) items to be accessed, \(\text{elements}(X)\) is the number of predefined datatypes in the typemap of \(X\), and \(old_{\text{file}}_{\text{offset}}\) is the value of the implicit offset before the call. The file position, \(new_{\text{file}}_{\text{offset}}\), is in terms of a count of etypes relative to the current view.

Synchronism

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 initiates an I/O operation, but does not wait for it to complete.

Given suitable hardware, this allows the transfer of data out/in the user’s buffer to proceed concurrently with computation. A separate request complete call (\text{MPI\_WAIT}, \text{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 \text{MPI\_FILE\_XXX}, where the I stands for immediate.

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 13.4.5, page 404).

Coordination

Every noncollective data access routine \text{MPI\_FILE\_XXX} has a collective counterpart. For most routines, this counterpart is \text{MPI\_FILE\_XXX\_ALL} or a pair of \text{MPI\_FILE\_XXX\_BEGIN} and \text{MPI\_FILE\_XXX\_END}. The counterparts to the \text{MPI\_FILE\_XXX\_SHARED} routines are \text{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 13.6.4, page 423, 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 \(\text{datatype}\).

Upon completion, the amount of data accessed by the calling process is returned in a \text{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 \(\text{offset}\) as an argument.
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 on page 27 and Section 4.1.11 on page 101. The data is accessed from those parts of the file specified by the current view (Section 13.3, page 385). 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 subsections “Problems Due to Data Copying and Sequence Association,” and “A Problem with Register Optimization” in Section 16.2.2, pages 463 and 466. (End of advice to users.)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS, respectively. The interpretation of the MPI_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE in the status argument if the return value of this argument is not needed. In C++, the status argument is optional. The status can be passed to MPI_TEST_CANCELED to determine if the operation was cancelled. All other fields of status are undefined.

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).

### 13.4.2 Data Access with Explicit Offsets

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section.

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections “Problems Due to Data Copying and Sequence Association,” and “A Problem with Register Optimization” in Section 16.2.2, pages 463 and 466. (End of advice to users.)

For blocking routines, status is returned directly. For nonblocking routines and split collective routines, status is returned when the operation is completed. The number of datatype entries and predefined elements accessed by the calling process can be extracted from status by using MPI_GET_COUNT and MPI_GET_ELEMENTS, respectively. The interpretation of the MPI_ERROR field is the same as for other operations — normally undefined, but meaningful if an MPI routine returns MPI_ERR_IN_STATUS. The user can pass (in C and Fortran) MPI_STATUS_IGNORE in the status argument if the return value of this argument is not needed. In C++, the status argument is optional. The status can be passed to MPI_TEST_CANCELED to determine if the operation was cancelled. All other fields of status are undefined.

When reading, a program can detect the end of file by noting that the amount of data read is less than the amount requested. Writing past the end of file increases the file size. The amount of data accessed will be the amount requested, unless an error is raised (or a read reaches the end of file).
13.4. DATA ACCESS

MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)

IN  fh    file handle (handle)
IN  offset file offset (integer)
OUT buf initial address of buffer (choice)
IN  count number of elements in buffer (integer)
IN  datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
<typename BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
void MPI::File::Read_at(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status)

MPI_FILE_READ_AT reads a file beginning at the position specified by offset.

MPI_FILE_READ_AT_ALL(fh, offset, buf, count, datatype, status)

IN  fh    file handle (handle)
IN  offset file offset (integer)
OUT buf initial address of buffer (choice)
IN  count number of elements in buffer (integer)
IN  datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
<typename BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
void MPI::File::Read_at_all(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status)

MPI_FILE_READ_AT_ALL reads a file beginning at the position specified by offset.
MPI_FILE_WRITE_AT_ALL is a collective version of the blocking MPI_FILE_WRITE_AT interface.

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)

INOUT fh file handle (handle)
INT offset file offset (integer)
INT buf initial address of buffer (choice)
INT count number of elements in buffer (integer)
INT datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
int MPI_Offset offset, void *buf, int count, const MPI::Datatype& datatype, MPI::Status& status)

MPI_FILE_WRITE_AT_ALL(fh, offset, buf, count, datatype, status)

INOUT fh file handle (handle)
INT offset file offset (integer)
INT buf initial address of buffer (choice)
INT count number of elements in buffer (integer)
INT datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf,
int MPI_Offset offset, void *buf, int count, const MPI::Datatype& datatype, MPI::Status& status)
13.4. DATA ACCESS

void MPI::File::Write_at_all(MPI::Offset offset, const void* buf, int count, const MPI::DataType& datatype)

MPI_FILE_WRITE_AT_ALL is a collective version of the blocking MPI_WRITE_AT interface.

MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)
IN fh file handle (handle)
IN offset file offset (integer)
OUT buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT request request object (handle)

int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
<type> BUF(*)INTEGER FH, COUNT, DATATYPE, REQUEST, IERRORINTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI::Request MPI::File::Iread_at(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype)

MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.

MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)
INOUT fh file handle (handle)
IN offset file offset (integer)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT request request object (handle)

int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
<type> BUF(*)INTEGER FH, COUNT, DATATYPE, REQUEST, IERRORINTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI::Request MPI::File::Iwrite_at(MPI::Offset offset, const void* buf, int count, const MPI::Datatype& datatype)
MPI_FILE_IWRITE_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.

### 13.4.3 Data Access with Individual File Pointers

MPI maintains one individual file pointer per process per file handle. The current value of this pointer implicitly specifies the offset in the data access routines described in this section. These routines only use and update the individual file pointers maintained by MPI. The shared file pointer is not used nor updated.

The individual file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 390, with the following modification:

- the offset is defined to be the current value of the MPI-maintained individual file pointer.

After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file.

- If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the routines in this section, with the exception of MPI_FILE_GET_BYTE_OFFSET.

\[
\text{MPI\_FILE\_READ}(fh, \text{buf, count, datatype, status})
\]

\[
\text{MPI\_FILE\_READ}(fh, \text{buf, count, datatype, status})
\]

\[
\text{MPI\_FILE\_READ}(\text{fh, buf, count, datatype, status})
\]

\[
\text{MPI\_FILE\_READ}(\text{fh, buf, count, datatype, status})
\]

\[
\text{MPI\_FILE\_READ}(\text{fh, buf, count, datatype, status})
\]

\[
\text{MPI\_FILE\_READ}(\text{fh, buf, count, datatype, status})
\]

\[
\text{MPI\_FILE\_READ}(\text{fh, buf, count, datatype, status})
\]

Example 13.2 The following Fortran code fragment is an example of reading a file until the end of file is reached:

```fortran
! 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.
```
integer bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
parameter (bufsize=100)
real localbuffer(bufsize)
call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
MPI_INFO_NULL, ierr )
totprocessed = 0
do
  call MPI_FILE_READ( myfh, localbuffer, bufsize, MPI_REAL, &
status, ierr )
call MPI_GET_COUNT( status, MPI_REAL, numread, ierr )
call process_input( localbuffer, numread )
totprocessed = totprocessed + numread
if ( numread < bufsize ) exit
enddo
write(6,1001) numread, bufsize, totprocessed
1001 format( 'No more data: read', I3, "and expected", I3, &
"Processed total of", I6, "before terminating job."
)
call MPI_FILE_CLOSE( myfh, ierr )

MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
INOUT fh file handle (handle)
OUT buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_read_all(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ_ALL(fh, buf, count, datatype, status, *status)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
void MPI::File::Read_all(void* buf, int count,
const MPI::Datatype& datatype, MPI::Status& status)
void MPI::File::Read_all(void* buf, int count,
const MPI::Datatype& datatype)

MPI_FILE_READ_ALL is a collective version of the blocking MPI_FILE_READ interface.
MPI_FILE_WRITE(fh, buf, count, datatype, status)

INTEGER fh
  file handle (handle)
INTEGER buf
  initial address of buffer (choice)
INTEGER count
  number of elements in buffer (integer)
INTEGER datatype
  datatype of each buffer element (handle)
INTEGER status
  status object (Status)

int MPI_File_write(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE(fh, buf, count, datatype, status)

MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)

INTEGER fh
  file handle (handle)
INTEGER buf
  initial address of buffer (choice)
INTEGER count
  number of elements in buffer (integer)
INTEGER datatype
  datatype of each buffer element (handle)
INTEGER status
  status object (Status)

int MPI_File_write_all(MPI_File fh, void *buf, int count,
                        MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)

MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE interface.
MPI_FILE_IREAD(fh, buf, count, datatype, request)

1. **INOUT** fh  
   file handle (handle)

2. **OUT** buf  
   initial address of buffer (choice)

3. **IN** count  
   number of elements in buffer (integer)

4. **IN** datatype  
   datatype of each buffer element (handle)

5. **OUT** request  
   request object (handle)

### C Function
```c
int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
```

### Fortran Function
```fortran
MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
```

### C++ Function
```cpp
MPI::Request MPI::File::Iread(void* buf, int count, const MPI::Datatype& datatype)
```

### Example 13.3
The following Fortran code fragment illustrates file pointer update semantics:

```fortran
integer bufsize, req1, req2
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
parameter (bufsize=10)
real buf1(bufsize), buf2(bufsize)
call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
   MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
   MPI_INFO_NULL, ierr )
call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
   req1, ierr )
call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
   req2, ierr )
call MPI_WAIT( req1, status1, ierr )
call MPI_WAIT( req2, status2, ierr )
call MPI_FILE_CLOSE( myfh, ierr )
```

### Example 13.3
The following Fortran code fragment illustrates file pointer update semantics:

```fortran
integer bufsize, req1, req2
integer, dimension(MPI_STATUS_SIZE) :: status1, status2
parameter (bufsize=10)
real buf1(bufsize), buf2(bufsize)
call MPI_FILE_OPEN( MPI_COMM_WORLD, 'myoldfile', &
   MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr )
call MPI_FILE_SET_VIEW( myfh, 0, MPI_REAL, MPI_REAL, 'native', &
   MPI_INFO_NULL, ierr )
call MPI_FILE_IREAD( myfh, buf1, bufsize, MPI_REAL, &
   req1, ierr )
call MPI_FILE_IREAD( myfh, buf2, bufsize, MPI_REAL, &
   req2, ierr )
call MPI_WAIT( req1, status1, ierr )
call MPI_WAIT( req2, status2, ierr )
call MPI_FILE_CLOSE( myfh, ierr )
```
MPI_FILE_IWRITE(fh, buf, count, datatype, request)

INOUT  fh  file handle (handle)
IN     buf  initial address of buffer (choice)
IN     count number of elements in buffer (integer)
IN     datatype datatype of each buffer element (handle)
OUT    request request object (handle)

int MPI_File_iwrite(MPI_File fh, void *buf, int count,
                     MPI_Datatype datatype, MPI_Request *request)

MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
<type> BUF(*)INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI::Request MPI::File::Iwrite(const void* buf, int count,
                               const MPI::Datatype& datatype)

MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.

MPI_FILE_SEEK(fh, offset, whence)

INOUT  fh  file handle (handle)
IN     offset file offset (integer)
IN     whence update mode (state)

int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)

MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
INTEGER FH, OFFSET, WHENCE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

void MPI::File::Seek(MPI::Offset offset, int whence)

MPI_FILE_SEEK updates the individual file pointer according to whence, which has the
following possible values:
• MPI_SEEK_SET: the pointer is set to offset
• MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
• MPI_SEEK_END: the pointer is set to the end of file plus offset

The offset can be negative, which allows seeking backwards. It is erroneous to seek to
a negative position in the view.

MPI_FILE_GET_POSITION(fh, offset)

IN     fh  file handle (handle)
OUT    offset offset of individual pointer (integer)

MPI_FILE_GET_POSITION(FH, OFFSET)
INTEGER FH, OFFSET
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_GET_POSITION updates the individual file pointer according to whence, which has the
following possible values:
• MPI_SEEK_SET: the pointer is set to offset
• MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
• MPI_SEEK_END: the pointer is set to the end of file plus offset

The offset can be negative, which allows seeking backwards. It is erroneous to seek to
a negative position in the view.
13.4. DATA ACCESS

MPITFile get_position(MPITFile fh, MPI Offset *offset) 

MPI_FILE_GET_POSITION(FH, OFFSET, IERROR) 
INTEGER FH, IERROR 
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET 
MPI::Offset MPI::File::Get_position() const 

MPI_FILE_GET_POSITION returns, in offset, the current position of the individual file pointere in etype units relative to the current view.

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.)

MPI_FILE_GET_BYTE_OFFSET (fh, offset, disp)

IN fh file handle (handle)
IN offset offset (integer)
OUT disp absolute byte position of offset (integer)

int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,(MPI_Offset *disp)

MPI_FILE_GET_BYTE_OFFSET (FH, OFFSET, DISP, IERROR) 
INTEGER FH, IERROR 
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP 
MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp) const 

MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte position. The absolute byte position (from the beginning of the file) of offset relative to the current view of fh is returned in disp.

13.4.4 Data Access with Shared File Pointers

MPI maintains exactly one shared file pointer per collective MPI OPEN (shared among processes in the communicator group). The current value of this pointer implicitly specifies the offset in the data access routines described in this section. These routines only use and update the shared file pointer maintained by MPI. The individual file pointers are not used nor updated.

The shared file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 390, with the following modifications:

• the offset is defined to be the current value of the MPI-maintained shared file pointer,
• the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and

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.)

MPI_FILE_GET_BYTE_OFFSET (fh, offset, disp)

IN fh file handle (handle)
IN offset offset (integer)
OUT disp absolute byte position of offset (integer)

int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,(MPI_Offset *disp)

MPI_FILE_GET_BYTE_OFFSET (FH, OFFSET, DISP, IERROR) 
INTEGER FH, IERROR 
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP 
MPI::Offset MPI::File::Get_byte_offset(const MPI::Offset disp) const 

MPI_FILE_GET_BYTE_OFFSET converts a view-relative offset into an absolute byte position. The absolute byte position (from the beginning of the file) of offset relative to the current view of fh is returned in disp.

13.4.4 Data Access with Shared File Pointers

MPI maintains exactly one shared file pointer per collective MPI OPEN (shared among processes in the communicator group). The current value of this pointer implicitly specifies the offset in the data access routines described in this section. These routines only use and update the shared file pointer maintained by MPI. The individual file pointers are not used nor updated.

The shared file pointer routines have the same semantics as the data access with explicit offset routines described in Section 13.4.2, page 390, with the following modifications:

• the offset is defined to be the current value of the MPI-maintained shared file pointer,
• the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and
the use of shared file pointer routines is erroneous unless all processes use the same file view.

For the noncollective shared file pointer routines, the serialization ordering is not deterministic. The user needs to use other synchronization means to enforce a specific order.

After a shared file pointer operation is initiated, the shared file pointer is updated to point to the next etype after the last one that will be accessed. The file pointer is updated relative to the current view of the file.

Noncollective Operations

MPI_FILE_READ_SHARED(fh, buf, count, datatype, status)

INOUT fh file handle (handle)
OUT buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<t>BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status)
void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype)

MPI_FILE_READ_SHARED reads a file using the shared file pointer.

MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)

INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_write_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<t>BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.

MPI_FILE_READ_SHARED(fh, buf, count, datatype, status)

INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<t>BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status)
void MPI::File::Read_shared(void* buf, int count, const MPI::Datatype& datatype)

MPI_FILE_READ_SHARED reads a file using the shared file pointer.

MPI_FILE_WRITE_SHARED(fh, buf, count, datatype, status)

INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

int MPI_File_write_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<t>BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Write_shared(void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status)
void MPI::File::Write_shared(void* buf, int count, const MPI::Datatype& datatype)

MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.
13.4. DATA ACCESS

INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Write_shared(const void* buf, int count,
   const MPI::Datatype& datatype, MPI::Status& status)

void MPI::File::Write_shared(const void* buf, int count,
   const MPI::Datatype& datatype)

MPI_FILE_WRITE_SHARED writes a file using the shared file pointer.

MPI_FILE_IREAD_SHARED(fh, buf, count, datatype, request)

INOUT fh file handle (handle)
OUT buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT request request object (handle)

int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
   MPI_Datatype datatype, MPI_Request *request)

MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)

MPI::Request MPI::File::Iread_shared(void* buf, int count,
   const MPI::Datatype& datatype)

MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED interface.

MPI_FILE_IWRITE_SHARED(fh, buf, count, datatype, request)

INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT request request object (handle)

int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
   MPI_Datatype datatype, MPI_Request *request)

MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)

MPI::Request MPI::File::Iwrite_shared(const void* buf, int count,
   const MPI::Datatype& datatype)

MPI_FILE_IWRITE_SHARED is a nonblocking version of the MPI_FILE_WRITE_SHARED interface.
MPI_FILE_IWRITE_SHARED is a nonblocking version of the MPI_FILE_WRITE_SHARED interface.

Collective Operations

The semantics of a collective access using a shared file pointer is that the accesses to the file will be in the order determined by the ranks of the processes within the group. For each process, the location in the file at which data is accessed is the position at which the shared file pointer would be after all processes whose ranks within the group less than that of this process had accessed their data. In addition, in order to prevent subsequent shared offset accesses by the same processes from interfering with this collective access, the call might return only after all the processes within the group have initiated their accesses. When the 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, count, datatype, status)

typedef struct {
    INOUT fh file handle (handle)
    OUT buf initial address of buffer (choice)
    IN count number of elements in buffer (integer)
    IN datatype datatype of each buffer element (handle)
    OUT status status object (Status)
} MPI_FILE_READ_ORDERED;

void MPI_File_read_ordered(MPI_File fh, void *buf, int count,
    const MPI_Datatype& datatype, MPI_Status& status)

MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

C interface:

int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

C interface:

int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

C interface:

int MPI_File_read_ordered(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

C interface:
MPI_FILE_READ_ORDERED is a collective version of the MPI_FILE_READ_SHARED interface.

MPI_FILE_WRITE_ORDERED(fh, buf, count, datatype, status)

| INOUT | fh       | file handle (handle) |
| IN    | buf      | initial address of buffer (choice) |
| IN    | count    | number of elements in buffer (integer) |
| OUT   | status   | status object (Status) |

int MPI_File_write_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)

void MPI::File::Write_ordered(const void* buf, int count, const MPI::Datatype& datatype, MPI::Status& status)

MPI_FILE_WRITE_ORDERED is a collective version of the MPI_FILE_WRITE_SHARED interface.

Seek

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call the following two routines (MPI_FILE_SEEK_SHARED and MPI_FILE_GET_POSITION_SHARED).

MPI_FILE_SEEK_SHARED(fh, offset, whence)

| INOUT  | fh       | file handle (handle) |
| IN     | offset   | file offset (integer) |
| IN     | whence   | update mode (state) |

int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)

MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)

void MPI::File::Seek_shared(MPI::Offset offset, int whence)

MPI_FILE_SEEK_SHARED updates the shared file pointer according to whence, which has the following possible values:
• MPI\_SEEK\_SET: the pointer is set to offset
• MPI\_SEEK\_CUR: the pointer is set to the current pointer position plus offset
• MPI\_SEEK\_END: the pointer is set to the end of file plus offset

MPI\_FILE\_SEEK\_SHARED is collective; all the processes in the communicator group associated with the file handle fh must call MPI\_FILE\_SEEK\_SHARED with the same values for offset and whence.

The offset can be negative, which allows seeking backwards. It is erroneous to seek to a negative position in the view.

MPI\_FILE\_GET\_POSITION\_SHARED(fh, offset)

\[\text{int MPI\_File\_get\_position\_shared(MPI\_File \text{ fh}, MPI\_Offset \text{ offset})}\]

MPI\_FILE\_GET\_POSITION\_SHARED(fh, offset, ierr)

\[\text{INTEGER}\ \text{fh, ierr}\]

\[\text{INTEGER(KIND=MPI\_OFFSET\_KIND) offset}\]

\[\text{MPI\::\File::Get\_position\_shared() const}\]

Advice to users. The offset can be used in a future call to MPI\_FILE\_SEEK\_SHARED 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.)

13.4.5 Split Collective Data Access Routines

MPI provides a restricted form of “nonblocking collective” I/O operations for all data accesses using split collective data access routines. These routines are referred to as “split” collective routines because a single collective operation is split in two: a begin routine and an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., MPI\_FILE\_IREAD). The end routine completes the operation, much like the matching test or wait (e.g., MPI\_WAIT). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc.

Split collective data access operations on a file handle fh are subject to the semantic rules given below.

• On any MPI process, each file handle may have at most one active split collective operation at any time.

MPI\_FILE\_GET\_POSITION\_SHARED(fh, offset)

\[\text{int MPI\_File\_get\_position\_shared(MPI\_File \text{ fh}, MPI\_Offset \text{ offset})}\]

MPI\_FILE\_GET\_POSITION\_SHARED(fh, offset, ierr)

\[\text{INTEGER}\ \text{fh, ierr}\]

\[\text{INTEGER(KIND=MPI\_OFFSET\_KIND) offset}\]

\[\text{MPI\::\File::Get\_position\_shared() const}\]

Advice to users. The offset can be used in a future call to MPI\_FILE\_SEEK\_SHARED 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.)

13.4.5 Split Collective Data Access Routines

MPI provides a restricted form of “nonblocking collective” I/O operations for all data accesses using split collective data access routines. These routines are referred to as “split” collective routines because a single collective operation is split in two: a begin routine and an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., MPI\_FILE\_IREAD). The end routine completes the operation, much like the matching test or wait (e.g., MPI\_WAIT). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc.

Split collective data access operations on a file handle fh are subject to the semantic rules given below.

• On any MPI process, each file handle may have at most one active split collective operation at any time.
13.4. DATA ACCESS

- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.

- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an “end” call is made, exactly one unmatched “begin” call for the same operation must precede it.

- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., `MPI_FILE_READ_ALL_BEGIN`) or the end call (e.g., `MPI_FILE_READ_ALL_END`) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.

- 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 many (though not all) of the problems described in “A Problem with Register Optimization,” Section 16.2.2, page 466.

- 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`).

- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.

- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an “end” call is made, exactly one unmatched “begin” call for the same operation must precede it.

- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., `MPI_FILE_READ_ALL_BEGIN`) or the end call (e.g., `MPI_FILE_READ_ALL_END`) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.

- 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 many (though not all) of the problems described in “A Problem with Register Optimization,” Section 16.2.2, page 466.

- 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 13.6.1, page 420), 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.

```c
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype)

MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
```

```c
void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf, int count, const MPI::Datatype& datatype)
```

```c
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)

MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
```

```c
void MPI::File::Read_at_all_end(void* buf, MPI::Status& status)
```
13.4. DATA ACCESS

**MPI_FILE_WRITE_AT_ALL_BEGIN**

```
INOUT fh  file handle (handle)
IN offset file offset (integer)
IN buf  initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
```

```c
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype)
```

### IN/OUT Parameters
- **fh**: The file handle (MPI_File).
- **offset**: The file offset (MPI_Offset).
- **buf**: The initial address of the buffer (void*).
- **count**: The number of elements in the buffer (int).
- **datatype**: The datatype of each buffer element (MPI_Datatype).

#### Description
This function writes all data at an offset in the file specified by the file handle `fh`. It takes the buffer `buf`, the offset `offset`, the number of elements `count`, and the datatype `datatype` as parameters.

### Example Usage
```
int status;
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, int *status);
```

**MPI_FILE_READ_AT_ALL_BEGIN**

```
INOUT fh  file handle (handle)
IN buf  initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
```

```c
int MPI_File_read_at_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)
```

### IN/OUT Parameters
- **fh**: The file handle (MPI_File).
- **buf**: The initial address of the buffer (void*).
- **count**: The number of elements in the buffer (int).
- **datatype**: The datatype of each buffer element (MPI_Datatype).

#### Description
This function reads all data at the offset in the file specified by the file handle `fh`. It takes the buffer `buf`, the number of elements `count`, and the datatype `datatype` as parameters.

### Example Usage
```
int status;
int MPI_File_read_at_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype, int *status);
```
void MPI::File::Read_all_begin(void* buf, int count, const MPI::Datatype& datatype) {
    MPI(FILE_READ_ALL_END(fh, buf, status))
}

int MPI_File_read_all_end(MPI_File fh, void* buf, MPI_Status* status) {
    MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
}

void MPI::File::Read_all_end(void* buf, MPI::Status& status) {
    void MPI::File::Read_all_end(void* buf)
}

MPI_FILE_WRITE_ALL_BEGIN(fh, count, datatype)

void MPI::File::Write_all_begin(const void* buf, int count, const MPI::Datatype& datatype) {
    MPI(FILE_WRITE_ALL_END(fh, buf, status))
}

int MPI_File_write_all_end(MPI_File fh, void* buf, MPI_Status* status) {
    MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
}
13.4. DATA ACCESS

MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
void MPI::File::Write_all_end(const void* buf, MPI::Status& status)
void MPI::File::Write_all_end(const void* buf)

MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
void MPI::File::Write_all_end(const void* buf, MPI::Status& status)
void MPI::File::Write_all_end(const void* buf)

MPI_FILE_READ_ORDERED_BEGIN(fh, buf, count, datatype)
INOUT fh  file handle (handle)
OUT  buf  initial address of buffer (choice)
IN  count number of elements in buffer (integer)
IN  datatype datatype of each buffer element (handle)

int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)

MPI_FILE_READ_ORDERED_BEGIN(fh, buf, COUNT, DATATYPE, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, IERROR
void MPI::File::Read_ordered_begin(void* buf, int count, const MPI::Datatype& datatype)

MPI_FILE_READ_ORDERED_END(fh, buf, status)
INOUT fh  file handle (handle)
OUT  buf  initial address of buffer (choice)
OUT status status object (Status)

int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)

MPI_FILE_READ_ORDERED_END(fh, buf, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
void MPI::File::Read_ordered_end(void* buf, MPI::Status& status)
void MPI::File::Read_ordered_end(void* buf)

MPI_FILE_READ_ORDERED_END(fh, buf, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
void MPI::File::Read_ordered_end(void* buf, MPI::Status& status)
void MPI::File::Read_ordered_end(void* buf)
MPI_FILE_WRITE_ORDERED_BEGIN(fh, buf, count, datatype)

INO fh  file handle (handle)
IN buf  initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)

int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)

MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
<type> BUF(*)INTEGER FH, COUNT, DATATYPE, IERROR

void MPI::File::Write_ordered_begin(const void* buf, int count, const MPI::Datatype& datatype)

MPI_FILE_WRITE_ORDERED_END(fh, buf, status)

INO fh  file handle (handle)
IN buf  initial address of buffer (choice)
OUT status status object (Status)

int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)

MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
<type> BUF(*) INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Write_ordered_end(const void* buf, MPI::Status& status)

void MPI::File::Write_ordered_end(const void* buf)

13.5 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 13.5.2, page 414) as well as the data conversion functions (Section 13.5.3, page 415).

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 13.6.1, page 420), 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 operations similar to POSIX cp, rm, and mv can be performed on the file. Furthermore, it is expected that the facility provided maintains the correspondence between absolute byte offsets (e.g., after possible file structure conversion, the data bits at byte offset 102 in the MPI environment are at byte offset 102 outside the MPI environment). As an example, a simple off-line conversion utility that transfers and converts files between the native file system and the MPI environment would suffice, provided it maintained the offset coherence mentioned above. In a high-quality implementation of MPI, users will be able to manipulate MPI files using the same or similar tools that the native file system offers for manipulating its files.

The remaining aspect of file interoperability, converting between different machine representations, is supported by the typing information specified in the etype and filetype. This facility allows the information in files to be shared between any two applications, regardless of whether they use MPI, and regardless of the machine architectures on which they run.

MPI supports multiple data representations: “native,” “internal,” and “external32.” An implementation may support additional data representations. MPI also supports user-defined data representations (see Section 13.5.3, page 415). The “native” and “internal” data representations are implementation dependent, while the “external32” representation is common to all MPI implementations and facilitates file interoperability. The data representation is specified 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.)*

**“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 data type conversions itself. *(End of advice to users.)*
CHAPTER 13. I/O

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.)

“internal” This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementation-defined and must be documented by the implementation.

Rationale. This data representation allows the implementation to perform I/O efficiently in a heterogeneous environment, though with implementation-defined restrictions on how the file can be reused. (End of rationale.)

Advice to implementors. Since “external32” is a superset of the functionality provided by “internal,” an implementation may choose to implement “internal” as “external32.” (End of advice to implementors.)

“external32” This data representation states that read and write operations convert all data from and to the “external32” representation defined in Section 13.5.2, page 414.

The data conversion rules for communication also apply to these conversions (see Section 3.3.2, page 25-27, of the MPI-1 document). 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 data type 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 data type conversions and the associated further loss of precision and performance. (End of advice to implementors.)

13.5.1 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

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.)

“internal” This data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementation-defined and must be documented by the implementation.

Rationale. This data representation allows the implementation to perform I/O efficiently in a heterogeneous environment, though with implementation-defined restrictions on how the file can be reused. (End of rationale.)

Advice to implementors. Since “external32” is a superset of the functionality provided by “internal,” an implementation may choose to implement “internal” as “external32.” (End of advice to implementors.)

“external32” This data representation states that read and write operations convert all data from and to the “external32” representation defined in Section 13.5.2, page 414.

The data conversion rules for communication also apply to these conversions (see Section 3.3.2, page 25-27, of the MPI-1 document). 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 data type 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 data type conversions and the associated further loss of precision and performance. (End of advice to implementors.)

13.5.1 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 filetype that are portable datatypes (see Section 2.4, page 11), 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.

Advice to users. One can logically think of the file as if it were stored in the memory of a file server. The etype and filetype are interpreted as if they were defined 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 the same architecture as the calling process, so that these types define the same data layout on the file as they would define in the memory of the calling process. If the etype 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. The routine MPI_FILE_GET_FILE_EXTENT can be used to calculate this scaling factor. Thus, two equivalent, portable datatypes will define the same data layout in the file, even in a heterogeneous environment with “internal”, “external”, or user-defined data representations. Otherwise, the etype and filetype must be constructed so that their typemap and extent are the same on any architecture. This can be achieved if they have an explicit upper bound and lower bound (defined either using MPI_LB and MPI_UB markers, or using MPI_TYPE_CREATE_RESIZED). This condition must also be fulfilled by any datatype that is used in the construction of the etype and filetype, if this datatype is replicated contiguously, either explicitly, by a call to MPI_TYPE_CONTIGUOUS, or implicitly, by a blocklength argument that is greater than one. If an etype or filetype is not portable, and has a typemap or extent that is architecture dependent, then the data layout specified by it on a file is implementation dependent.

File data representations other than “native” may be different from corresponding data representations in memory. Therefore, for these file data representations, it is important not to use hardwired byte offsets for file positioning, including the initial displacement that specifies the view. When a portable datatype (see Section 2.4, page 11) is used in a data access operation, any holes in the datatype are scaled to match the data representation. However, note that this technique only works when all the processes that created the file view build their etypes from the same predefined datatypes. For example, if one process uses an etype built from MPI_INT and another uses an etype built from MPI_FLOAT, the resulting views may be nonportable because the relative sizes of these types may differ from one data representation to another. (End of advice to users.)

MPI_FILE_GET_TYPE_EXTENT(fh, datatype, extent)

<table>
<thead>
<tr>
<th>IN</th>
<th>fh</th>
<th>file handle (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>extent</td>
<td>datatype extent (integer)</td>
</tr>
</tbody>
</table>

MPI_FILE_GET_FILE_EXTENT(fh, datatype, extent)

<table>
<thead>
<tr>
<th>IN</th>
<th>fh</th>
<th>file handle (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>extent</td>
<td>datatype extent (integer)</td>
</tr>
</tbody>
</table>
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype, MPI_Aint *extent)

MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)

INTEGER FH, DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT

MPI::Aint MPI::File::Get_type_extent(const MPI::Datatype& datatype) const

in the file. 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 13.5.3, page 415), MPI uses the dtype_file_extent_fn callback to calculate the extent.

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 13.5.3, page 415). (End of advice to implementors.)

13.5.2 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 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 = +16383, 112 fraction bits, and an encoding analogous to the “Double” format. All integral values are in two’s complement big-endian format. Big-endian means most significant byte at lowest address byte. For Fortran LOGICAL and C++ bool, 0 implies false and nonzero implies true. Fortran COMPLEX and DOUBLE COMPLEX are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [28]. Wide characters (of type MPI_WCHAR) are in Unicode format [47].

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 [27], 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 (see ftp://ds.internic.net/rfc/rfc1832.txt). (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.)
The size of the predefined datatypes returned from `MPI_TYPE_CREATE_F90_REAL`, `MPI_TYPE_CREATE_F90_COMPLEX`, and `MPI_TYPE_CREATE_F90_INTEGER` are defined in Section 16.2.5, page 474.

**Advice to implementors.** When converting a larger size integer to a smaller size integer, only the less significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (End of advice to implementors.)

Table 13.2 specifies the sizes of predefined datatypes in “external32” format.

### 13.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

1. a user wants to write a file in a representation unknown to the implementation, and
2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.

```c
MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn, dtype_file_extent_fn, extra_state)
```

**IN**
- `datarep` data representation identifier (string)
- `read_conversion_fn` function invoked to convert from file representation to native representation (function)
- `write_conversion_fn` function invoked to convert from native representation to file representation (function)
- `dtype_file_extent_fn` function invoked to get the extent of a datatype as represented in the file (function)
- `extra_state` extra state

```c
int MPI_Register_datarep(char *datarep, MPI_Datarep_conversion_function *read_conversion_fn, MPI_Datarep_conversion_function *write_conversion_fn, MPI_DatarepExtent_function *dtype_file_extent_fn, void *extra_state)
```

```c
void MPI::Register_datarep(const char* datarep, MPI::Datarep_conversion_function* read_conversion_fn, MPI::Datarep_conversion_function* write_conversion_fn, MPI::DatarepExtent_function* dtype_file_extent_fn, extra_state),
```

Advice to implementors. When converting a larger size integer to a smaller size integer, only the less significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (End of advice to implementors.)

Table 13.2 specifies the sizes of predefined datatypes in “external32” format.

### 13.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

1. a user wants to write a file in a representation unknown to the implementation, and
2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.

```c
MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn, dtype_file_extent_fn, extra_state)
```

**IN**
- `datarep` data representation identifier (string)
- `read_conversion_fn` function invoked to convert from file representation to native representation (function)
- `write_conversion_fn` function invoked to convert from native representation to file representation (function)
- `dtype_file_extent_fn` function invoked to get the extent of a datatype as represented in the file (function)
- `extra_state` extra state

Advice to implementors. When converting a larger size integer to a smaller size integer, only the less significant bytes are moved. Care must be taken to preserve the sign bit value. This allows no conversion errors if the data range is within the range of the smaller size integer. (End of advice to implementors.)

Table 13.2 specifies the sizes of predefined datatypes in “external32” format.

### 13.5.3 User-Defined Data Representations

There are two situations that cannot be handled by the required representations:

1. a user wants to write a file in a representation unknown to the implementation, and
2. a user wants to read a file written in a representation unknown to the implementation.

User-defined data representations allow the user to insert a third party converter into the I/O stream to do the data representation conversion.
<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_PACKED</td>
<td>1</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>1</td>
</tr>
<tr>
<td>MPI_CHAR</td>
<td>1</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>1</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>1</td>
</tr>
<tr>
<td>MPI_WCHAR</td>
<td>2</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>2</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>2</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>4</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>4</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>4</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>4</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
<td>8</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>8</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>4</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>8</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>16</td>
</tr>
<tr>
<td>MPI_INTEGER1</td>
<td>1</td>
</tr>
<tr>
<td>MPI_INTEGER2</td>
<td>2</td>
</tr>
<tr>
<td>MPI_INTEGER4</td>
<td>4</td>
</tr>
<tr>
<td>MPI_INTEGER8</td>
<td>8</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>4</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>8</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>2*4</td>
</tr>
<tr>
<td>MPI_DOUBLE_COMPLEX</td>
<td>2*8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER1</td>
<td>1</td>
</tr>
<tr>
<td>MPI_INTEGER2</td>
<td>2</td>
</tr>
<tr>
<td>MPI_INTEGER4</td>
<td>4</td>
</tr>
<tr>
<td>MPI_INTEGER8</td>
<td>8</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>4</td>
</tr>
<tr>
<td>MPI_REAL8</td>
<td>8</td>
</tr>
<tr>
<td>MPI_REAL16</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 13.2: “external32” sizes of predefined datatypes
Datarep Conversion Functions

typedef int MPI_Datarep_conversion_function(MPI_Datatype datatype,
MPI_Aint *userbuf, void *extra_state);

SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
MIPI_OFFSET position, void *extra_state);

typedef void MPI_Datarep_conversion_function(void *userbuf,
const MPI::Datatype& datatype, int count, void *filebuf,
MPI::Offset position, void *extra_state);

The function read_conversion_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with

datarep

The call associates read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn with the data representation identifier datarep. datarep can then be used as an argument to MPI_FILE_SET_VIEW, causing subsequent data access operations to call the conversion functions to convert all data items accessed between file data representation and native 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 13.7, page 429). 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

typedef int MPI_Datarep_extent_function(const MPI::Datatype& datatype,
MPI_Aint *file_extent, void *extra_state);

SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

typedef void MPI::Datarep_extent_function(const MPI::Datatype& datatype,
MPI::Aint& file_extent, void* extra_state);

The function dtype_file_extent_fn must return, in file_extent, the number of bytes required to store datatype in the file representation. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call this routine with predefined datatypes employed by the user.

Datarep Conversion Functions

typedef int MPI_Datarep_conversion_function(void *userbuf,
MPI_Datatype datatype, int count, void *filebuf,
MPI_Offset position, void *extra_state);

SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
POSITION, EXTRA_STATE, IERROR)
(TYPE) USERBUF(*), FILEBUF(*)
INTEGER COUNT, DATATYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) POSITION
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

typedef void MPI::Datarep_conversion_function(void* userbuf,
MPI::Datatype& datatype, int count, void* filebuf,
MPI::Offset position, void* extra_state);

The function read_conversion_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with

Extent Callback

typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
MPI_Aint *file_extent, void *extra_state);

SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

typedef void MPI::Datarep_extent_function(const MPI::Datatype& datatype,
MPI::Aint& file_extent, void* extra_state);

The function dtype_file_extent_fn must return, in file_extent, the number of bytes required to store datatype in the file representation. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call this routine with predefined datatypes employed by the user.

Datarep Conversion Functions

typedef int MPI_Datarep_conversion_function(void *userbuf,
MPI_Datatype datatype, int count, void *filebuf,
MPI_Offset position, void *extra_state);

SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
POSITION, EXTRA_STATE, IERROR)
(TYPE) USERBUF(*), FILEBUF(*)
INTEGER COUNT, DATATYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) POSITION
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

typedef void MPI::Datarep_conversion_function(void* userbuf,
MPI::Datatype& datatype, int count, void* filebuf,
MPI::Offset position, void* extra_state);

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 contingously 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 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.)
The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold 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 must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

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 that case, 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 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 dttype_file_extent_fn) when one of the read or write routines in Section 13.4, page 387, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dttype_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.

### 13.5.4 Matching Data Representations

It is the user’s responsibility to ensure that the data representation used to read data from a file is compatible with the data representation that was used to write that data to the file. 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 13.5.2, page 414, 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.

The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold 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 must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

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 that case, 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 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 dttype_file_extent_fn) when one of the read or write routines in Section 13.4, page 387, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dttype_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.

### 13.5.4 Matching Data Representations

It is the user’s responsibility to ensure that the data representation used to read data from a file is compatible with the data representation that was used to write that data to the file. 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 13.5.2, page 414, 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 16.2.5, page 470).

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 16.2.5, page 470, defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

13.6 Consistency and Semantics

13.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file 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 other than the above. Sequential consistency means the behavior of a set of operations will be 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. User-imposed 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 file $FOO$, and $FH_2$ be the set of file handles created from a different collective open of $FOO$. Note that nothing restrictive is said about $FH_1$ and $FH_2$: the sizes of $FH_1$ and $FH_2$ may be different, the groups of processes used for each open may or may not intersect, the file handles in $FH_1$ may be destroyed before those in $FH_2$ are created, etc. Consider the following three cases: a single file handle ($e.g., fh_1 \in FH_1$), two file handles created from a single collective open ($e.g., fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from different collective opens ($e.g., fh_1 \in FH_1$ and $fh_2 \in FH_2$).

For the purpose of consistency semantics, a matched pair (Section 13.4.5, page 404) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and 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

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 16.2.5, page 470).

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 16.2.5, page 470, defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (End of advice to users.)

13.6 Consistency and Semantics

13.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file 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 other than the above. Sequential consistency means the behavior of a set of operations will be 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. User-imposed 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 file $FOO$, and $FH_2$ be the set of file handles created from a different collective open of $FOO$. Note that nothing restrictive is said about $FH_1$ and $FH_2$: the sizes of $FH_1$ and $FH_2$ may be different, the groups of processes used for each open may or may not intersect, the file handles in $FH_1$ may be destroyed before those in $FH_2$ are created, etc. Consider the following three cases: a single file handle ($e.g., fh_1 \in FH_1$), two file handles created from a single collective open ($e.g., fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from different collective opens ($e.g., fh_1 \in FH_1$ and $fh_2 \in FH_2$).

For the purpose of consistency semantics, a matched pair (Section 13.4.5, page 404) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and 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 $SEQ_{ja}$ be a sequence of file operations on a single file handle, bracketed by MPI_FILE_SYNC() on that file handle. (Both opening and closing a file implicitly perform an MPI_FILE_SYNC.) $SEQ_{ja}$ is a “write sequence” if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two sequences, $SEQ_1$ and $SEQ_2$, we say they are not concurrent if one sequence is guaranteed 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.

Case 1: $fh_1 \in FH_1$ All operations on $fh_1$ are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on $fh_1$ are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_1a \in FH_1$ and $fh_1b \in FH_1$ Assume $A_1$ is a data access operation using $fh_1a$, and $A_2$ is a data access operation using $fh_1b$. If for any access $A_1$, there is no access $A_2$ that conflicts with $A_1$, then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If $A_1$ and $A_2$ conflict, sequential consistency can be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, or meeting the condition described in Case 3 below.

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).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence $SEQ_1$ to the file, there is no sequence $SEQ_2$ to the file which is concurrent with $SEQ_1$. To guarantee sequential consistency when there are write sequences, MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 13.6.10, page 425, for further clarification of some of these consistency semantics.

MPI_FILE_SET_ATOMICITY$(fh, flag)$

INOUT $fh$ file handle (handle)

IN $flag$ true to set atomic mode, false to set nonatomic mode (logical)

int MPI_File_set_atomicity(MPI_File fh, int flag)

MPI_FILE_SET_ATOMICITY$(FH, FLAG, IERROR)$

INTEGER FH, IERROR

LOGICAL FLAG

13.6. CONSISTENCY AND SEMANTICS

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 $SEQ_{ja}$ be a sequence of file operations on a single file handle, bracketed by MPI_FILE_SYNC() on that file handle. (Both opening and closing a file implicitly perform an MPI_FILE_SYNC.) $SEQ_{ja}$ is a “write sequence” if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two sequences, $SEQ_1$ and $SEQ_2$, we say they are not concurrent if one sequence is guaranteed 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.

Case 1: $fh_1 \in FH_1$ All operations on $fh_1$ are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on $fh_1$ are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_1a \in FH_1$ and $fh_1b \in FH_1$ Assume $A_1$ is a data access operation using $fh_1a$, and $A_2$ is a data access operation using $fh_1b$. If for any access $A_1$, there is no access $A_2$ that conflicts with $A_1$, then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If $A_1$ and $A_2$ conflict, sequential consistency can be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, or meeting the condition described in Case 3 below.

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).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence $SEQ_1$ to the file, there is no sequence $SEQ_2$ to the file which is concurrent with $SEQ_1$. To guarantee sequential consistency when there are write sequences, MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 13.6.10, page 425, for further clarification of some of these consistency semantics.
Let $\mathcal{F}_h$ be the set of file handles created by one collective open. The consistency semantics for data access operations using $\mathcal{F}_h$ is set by collectively calling 
\texttt{MPI\_FILE\_SET\_ATOMICITY} on $\mathcal{F}_h$. \texttt{MPI\_FILE\_SET\_ATOMICITY} is collective; all processes in the group must pass identical values for $\text{fh}$ and $\text{flag}$. If $\text{flag}$ is true, atomic mode is set; if $\text{flag}$ is false, nonatomic mode is set.

 Changing the consistency semantics for an open file only affects new data accesses. 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 not completed (e.g., via \texttt{MPI\_WAIT}) are only guaranteed to abide by nonatomic mode consistency semantics.

\textit{Advice to implementors.} Since the semantics guaranteed by atomic mode are stronger than those guaranteed by nonatomic mode, an implementation is free to adhere to the more stringent atomic mode semantics for outstanding requests. (End of advice to implementors.)

\begin{verbatim}
void MPI::File::Set_atomicity(bool flag)

    Let \( \mathcal{F}_h \) be the set of file handles created by one collective open. The consistency semantics for data access operations using \( \mathcal{F}_h \) is set by collectively calling \texttt{MPI\_FILE\_SET\_ATOMICITY} on \( \mathcal{F}_h \). \texttt{MPI\_FILE\_SET\_ATOMICITY} is collective; all processes in the group must pass identical values for \text{fh} and \text{flag}. If \text{flag} is true, atomic mode is set; if \text{flag} is false, nonatomic mode is set.

    Changing the consistency semantics for an open file only affects new data accesses. 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 not completed (e.g., via \texttt{MPI\_WAIT}) are only guaranteed to abide by nonatomic mode consistency semantics.

    \textit{Advice to implementors.} Since the semantics guaranteed by atomic mode are stronger than those guaranteed by nonatomic mode, an implementation is free to adhere to the more stringent atomic mode semantics for outstanding requests. (End of advice to implementors.)

\end{verbatim}

MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see above).

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.

13.6.2 Random Access vs. Sequential Files

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.

Rationale. 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. (End of rationale.)

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.

13.6.3 Progress

The progress rules of MPI are both a promise to users and a set of constraints on implementations. 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.

13.6.4 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 5.12 on page 177.
Collective file operations are collective over a dup 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.

13.6.5 Type Matching
The type matching rules for I/O mimic the type matching rules for communication with one 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.

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.)

13.6.6 Miscellaneous Clarifications
Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI_FILE_SET_VIEW, and the datatype must be committed before calling MPI_FILE_READ or MPI_FILE_WRITE.

13.6.7 MPI_Offset Type
MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset.

In Fortran, the corresponding integer is an integer of kind MPI_OFFSET_KIND, defined in mpif.h and the mpi module.

In Fortran 77 environments that do not support KIND parameters, MPI_Offset arguments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 16.3, page 478).

13.6.8 Logical vs. Physical File Layout
MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as hints specified via info when a file is created (see Section 13.2.8, page 382).

Collective file operations are collective over a dup 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.

13.6.5 Type Matching
The type matching rules for I/O mimic the type matching rules for communication with one 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.

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.)

13.6.6 Miscellaneous Clarifications
Once an I/O routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the comm and info used in an MPI_FILE_OPEN, or the etype and filetype used in an MPI_FILE_SET_VIEW, can be freed without affecting access to the file. Note that for nonblocking routines and split collective operations, the operation must be completed before it is safe to reuse data buffers passed as arguments.

As in communication, datatypes must be committed before they can be used in file manipulation or data access operations. For example, the etype and filetype must be committed before calling MPI_FILE_SET_VIEW, and the datatype must be committed before calling MPI_FILE_READ or MPI_FILE_WRITE.

13.6.7 MPI_Offset Type
MPI_Offset is an integer type of size sufficient to represent the size (in bytes) of the largest file supported by MPI. Displacements and offsets are always specified as values of type MPI_Offset.

In Fortran, the corresponding integer is an integer of kind MPI_OFFSET_KIND, defined in mpif.h and the mpi module.

In Fortran 77 environments that do not support KIND parameters, MPI_Offset arguments should be declared as an INTEGER of suitable size. The language interoperability implications for MPI_Offset are similar to those for addresses (see Section 16.3, page 478).

13.6.8 Logical vs. Physical File Layout
MPI specifies how the data should be laid out in a virtual file structure (the view), not how that file structure is to be stored on one or more disks. Specification of the physical file structure was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information may be necessary to optimize file layout. This information can be provided as hints specified via info when a file is created (see Section 13.2.8, page 382).
13.6. CONSISTENCY AND SEMANTICS

13.6.9 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 13.6.1, page 420, 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.)

13.6.10 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.

```c
/* Process 0 */
int i, a[10];
int TRUE = 1;
```
for (i=0;i<10;i++)
a[i] = 5;

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh0, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

for (i=0;i<10;i++)
a[i] = 5;

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh1, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
temporal order with, for example, calls to MPI_BARRIER.

Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
received by process 1 using MPI_RECV. (End of advice to users.)

Alternatively, a user can impose consistency with nonatomic mode set:

for (i=0;i<10;i++)
a[i] = 5;

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh1, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh0, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh1, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
temporal order with, for example, calls to MPI_BARRIER.

Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
received by process 1 using MPI_RECV. (End of advice to users.)

Alternatively, a user can impose consistency with nonatomic mode set:

for (i=0;i<10;i++)
a[i] = 5;

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
MPI_File_sync( fh0 );
MPI_Barrier( MPI_COMM_WORLD );
MPI_File_sync( fh0 );

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh1, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
MPI_File_sync( fh0 );
MPI_Barrier( MPI_COMM_WORLD );
MPI_File_sync( fh0 );

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh1, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
MPI_File_sync( fh0 );
MPI_Barrier( MPI_COMM_WORLD );
MPI_File_sync( fh0 );

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 );
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_set_atomicity( fh1, TRUE );
/* MPI_Barrier( MPI_COMM_WORLD ); */

MPI_File_open( MPI_COMM_WORLD, "workfile", 
    MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 );
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL );
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status );
MPI_File_sync( fh0 );
MPI_Barrier( MPI_COMM_WORLD );
MPI_File_sync( fh0 );

A user may guarantee that the write on process 0 precedes the read on process 1 by imposing
temporal order with, for example, calls to MPI_BARRIER.

Advice to users. Routines other than MPI_BARRIER may be used to impose temporal
order. In the example above, process 0 could use MPI_SEND to send a 0 byte message,
received by process 1 using MPI_RECV. (End of advice to users.)

Alternatively, a user can impose consistency with nonatomic mode set:
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;

The “sync-barrier-sync” construct is required because:

• The barrier ensures that the write on process 0 occurs before the read on process 1.
• The first sync guarantees that the data written by all processes is transferred to the
  storage device.
• The second sync guarantees that all data which has been transferred to the storage
device is visible to all processes. (This does not affect process 0 in this example.)

The following program represents an erroneous attempt to achieve consistency by elim-
inating the apparently superfluous second “sync” call for each process.

/* ---------------- THIS EXAMPLE IS ERRONEOUS --------------- */
/* Process 0 */
int i, a[10] ;
for ( i=0;i<10;i++)
  a[i] = 5 ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
/* Process 1 */
int b[10] ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
/* ---------------- THIS EXAMPLE IS ERRONEOUS --------------- */

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.

Advice to users. Some implementations may choose to implement MPI_FILE_SYNC as a temporarily synchronizing function. When using such an implementation, the
“sync-barrier-sync” construct above can be replaced by a single “sync.” The results of
using such code with an implementation for which MPI_FILE_SYNC is not temporally
synchronizing is undefined. (End of advice to users.)

MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;

The “sync-barrier-sync” construct is required because:

• The barrier ensures that the write on process 0 occurs before the read on process 1.
• The first sync guarantees that the data written by all processes is transferred to the
  storage device.
• The second sync guarantees that all data which has been transferred to the storage
device is visible to all processes. (This does not affect process 0 in this example.)

The following program represents an erroneous attempt to achieve consistency by elim-
inating the apparently superfluous second “sync” call for each process.

/* ---------------- THIS EXAMPLE IS ERRONEOUS --------------- */
/* Process 0 */
int i, a[10] ;
for ( i=0;i<10;i++)
  a[i] = 5 ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0 ) ;
MPI_File_set_view( fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status ) ;
MPI_File_sync( fh0 ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
/* Process 1 */
int b[10] ;
MPI_File_open( MPI_COMM_WORLD, "workfile",
MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1 ) ;
MPI_File_set_view( fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_Barrier( MPI_COMM_WORLD ) ;
MPI_File_sync( fh1 ) ;
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status ) ;
/* ---------------- THIS EXAMPLE IS ERRONEOUS --------------- */

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.

Advice to users. Some implementations may choose to implement MPI_FILE_SYNC as a temporarily synchronizing function. When using such an implementation, the
“sync-barrier-sync” construct above can be replaced by a single “sync.” The results of
using such code with an implementation for which MPI_FILE_SYNC is not temporally
synchronizing is undefined. (End of advice to users.)
Asynchronous I/O

The behavior of asynchronous I/O operations is determined by applying the rules specified above for synchronous I/O operations.

The following examples all access a preexisting file “myfile.” Word 10 in myfile initially contains the integer 2. Each example writes and reads word 10.

First consider the following code fragment:

```c
int a = 4, b, TRUE=1;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Waitall(2, reqs, statuses) ;
```

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```c
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

```c
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
```

defines the same ordering as:

```c
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
```

defines the same ordering as:

```c
int a = 4, b, TRUE=1;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh, TRUE ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Waitall(2, reqs, statuses) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
FMLIPI
```

For asynchronous data access operations, MPI specifies that the access occurs at any time between the call to the asynchronous data access routine and the return from the corresponding request complete routine. Thus, executing either the read before the write, or the write before the read is consistent with program order. If atomic mode is set, then MPI guarantees sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```c
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
/* MPI_File_set_atomicity( fh, TRUE ) ; Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_Wait(&reqs[1], &status) ;
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

```c
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[1], &status) ;
```

defines the same ordering as:

```c
int a = 4, b, TRUE=1;
MPI_File_open( MPI_COMM_WORLD, "myfile",
               MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_set_atomicity( fh, TRUE ) ;
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]) ;
MPI_File_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_Wait(&reqs[1], &status) ;
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.
int a = 4, b;
MPI_File_open( MPI_COMM_WORLD, "myfile",
MPI_MODE_RDWR, MPI_INFO_NULL, &fh ) ;
MPI_File_set_view( fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL ) ;
MPI_File_write_at(fh, 10, &a, 1, MPI_INT, &status ) ;MPI_File_read_at(fh, 10, &b, 1, MPI_INT, &status ) ;

Since

• nonconcurrent operations on a single file handle are sequentially consistent, and

• the program fragments specify an order for the operations,

MPI guarantees that both program fragments will read the value 4 into b. There is no need to set atomic mode for this example.

Similar considerations apply to conflicting accesses of the form:

MPI_File_write_all_begin(fh,...) ;MPI_File_iread(fh,...) ;MPI_Wait(fh,...) ;MPI_File_write_all_end(fh,...) ;

Recall that constraints governing consistency and semantics are not relevant to the following:

MPI_File_write_all_begin(fh,...) ;MPI_File_read_all_begin(fh,...) ;MPI_File_read_all_end(fh,...) ;MPI_File_write_all_end(fh,...) ;

since split collective operations on the same file handle may not overlap (see Section 13.4.5, page 404).

13.7 I/O Error Handling

By default, communication errors are fatal—MPI_ERRORS_ARE_FATAL is the default error handler associated with MPI_COMM_WORLD. I/O errors are usually less catastrophic (e.g., “file not found”) than communication errors, and common practice is to catch these errors and continue executing. For this reason, MPI provides additional error facilities for I/O.

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 8.3, page 264.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in
MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL.

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN.

The default file error handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default 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 default 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 be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER.

Rationale. For communication, the default error handler is inherited from MPI_COMM_WORLD. 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.)

13.8 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 13.3.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

13.9 Examples

13.9.1 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().

/*=========================================================================
 */

* Function: double_buffer
* Synopsis: void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount )
* Description: Performs the steps to overlap computation with a collective write
* by using a double-buffering technique.

MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL.

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN.

The default file error handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default 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 default 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 be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER.

Rationale. For communication, the default error handler is inherited from MPI_COMM_WORLD. 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.)

13.8 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 13.3.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

13.9 Examples

13.9.1 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().

/*=========================================================================
 */

* Function: double_buffer
* Synopsis: void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount )
* Description: Performs the steps to overlap computation with a collective write
* by using a double-buffering technique.

/*=========================================================================
 */

* Function: double_buffer
* Synopsis: void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount )
* Description: Performs the steps to overlap computation with a collective write
* by using a double-buffering technique.

/*=========================================================================
 */

* Function: double_buffer
* Synopsis: void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount )
* Description: Performs the steps to overlap computation with a collective write
* by using a double-buffering technique.
13.9. EXAMPLES

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERR_FILE</td>
<td>Invalid file handle</td>
</tr>
<tr>
<td>MPI_ERR_NOT_SAME</td>
<td>Collective argument not identical on all processes, or collective routines called in a different order by different processes</td>
</tr>
<tr>
<td>MPI_ERR_AMODE</td>
<td>Error related to the amode passed to MPI_FILE_OPEN</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_DATAREP</td>
<td>Unsupported datarep passed to MPI_FILE_SET_VIEW</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_OPERATION</td>
<td>Unsupported operation, such as seeking on a file which supports sequential access only</td>
</tr>
<tr>
<td>MPI_ERR_NO_SUCH_FILE</td>
<td>File does not exist</td>
</tr>
<tr>
<td>MPI_ERR_FILE_EXISTS</td>
<td>File exists</td>
</tr>
<tr>
<td>MPI_ERR_BAD_FILE</td>
<td>Invalid file name (e.g., path name too long)</td>
</tr>
<tr>
<td>MPI_ERR_ACCESS</td>
<td>Permission denied</td>
</tr>
<tr>
<td>MPI_ERR_NO_SPACE</td>
<td>Not enough space</td>
</tr>
<tr>
<td>MPI_ERR_QUOTA</td>
<td>Quota exceeded</td>
</tr>
<tr>
<td>MPI_ERR_READ_ONLY</td>
<td>Read-only file or file system</td>
</tr>
<tr>
<td>MPI_ERR_FILE_IN_USE</td>
<td>File operation could not be completed, as the file is currently open by some process</td>
</tr>
<tr>
<td>MPI_ERR_DUP_DATAREP</td>
<td>Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP</td>
</tr>
<tr>
<td>MPI_ERR_CONVERSION</td>
<td>An error occurred in a user supplied data conversion function.</td>
</tr>
<tr>
<td>MPI_ERR_IO</td>
<td>Other I/O error</td>
</tr>
</tbody>
</table>

Table 13.3: I/O Error Classes
void double_buffer( MPI_File fh, MPI_Datatype buftype, int bufcount) {

    MPI_Status status; /* status for MPI calls */
    float *buffer1, *buffer2; /* buffers to hold results */
    float *compute_buf_ptr; /* destination buffer */
    /* for computing */
    float *write_buf_ptr; /* source for writing */
    int done; /* determines when to quit */
    /* buffer initialization */
    buffer1 = (float *) malloc(bufcount*sizeof(float));
    buffer2 = (float *) malloc(bufcount*sizeof(float));
    compute_buf_ptr = buffer1; /* initially point to buffer1 */
    write_buf_ptr = buffer1; /* initially point to buffer1 */

    /* DOUBLE-BUFFER prolog:
     * compute buffer1; then initiate writing buffer1 to disk */
    compute_buffer(compute_buf_ptr, bufcount, &done);
    MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);

    /* DOUBLE-BUFFER steady state:
     * Overlap writing old results from buffer pointed to by write_buf_ptr
     * 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, bufcount, buftype);
        TOGGLE_PTR(write_buf_ptr);
        MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
    }
}

#define TOGGLE_PTR(x) ((x)==(buffer1)) ? (x=buffer2) : (x=buffer1)
13.9. EXAMPLES

13.9.2 Subarray Filetype Constructor

Assume we are writing out a 100x100 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 13.4). To create the filetypes for each process one could use the following C program (see Section 4.1.3 on page 87):

```c
/* 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);
```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
MPI_DOUBLE, &filetype);

Or, equivalently in Fortran:

double precision subarray(100,25)
integer filetype, rank, ierror
integer sizes(2), subsizes(2), starts(2)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
sizes(1)=100
sizes(2)=100
subsizes(1)=100
subsizes(2)=25
starts(1)=0
starts(2)=rank*subsizes(2)

call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION, &
filetype, ierror)

The generated filetype will then describe the portion of the file contained within the
process's subarray with holes for the space taken by the other processes. Figure 13.5 shows
the filetype created for process 1.
Chapter 14

Profiling Interface

14.1 Requirements

To meet the MPI profiling interface, an implementation of the MPI functions must

1. provide a mechanism through which all of the MPI defined functions except those allowed as macros (See Section 2.6.5). This requires, in C and Fortran, an alternate entry point name, with the prefix PMPI_, for each MPI function. The profiling interface in C++ is described in Section 16.1.10. 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 MPL_ version with a user-defined version.

2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.

3. 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 she must implement the profile interface for each binding, or can economise by implementing it only for 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.

5. provide a no-op routine MPI_PCONTROL in the MPI library.

14.2 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.

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 its 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 chapter 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).

14.3 Logic of the Design

Provided that an MPI implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept all of the MPI calls that are made by the user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

14.3.1 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 is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation
- Adding user events to a trace file.

These requirements are met by use of the MPI_PCONTROL.
14.4. EXAMPLES

MPI_PCONTROL(level, ...) IN level Profiling level

int MPI_Pcontrol(const int level, ...)
MPI_PCONTROL(LEVEL)
INTEGER LEVEL, ...
void MPI::Pcontrol(const int 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. (This 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_INIT has been called is for profiling to be enabled at the normal default level. (i.e. as if MPI_PCONTROL had just been called with the argument 1). This allows users to link with a profiling library and obtain profile output without having to modify their source code at all.

The provision of MPI_PCONTROL as a no-op in the standard MPI library allows them to modify their source code to obtain more detailed profiling information, but still be able to link exactly the same code against the standard MPI library.

14.4 Examples

14.4.1 Profiler Implementation

Suppose that the profiler wishes to accumulate the total amount of data sent by the MPI_SEND function, along with the total elapsed time spent in the function. This could trivially be achieved thus

```c
static int totalBytes;
static double totalTime;

int MPI_SEND(void * buffer, const int count, MPI_Datatype datatype,
              int dest, int tag, MPI_comm comm)
{
    double tstart = MPI_Wtime(); /* Pass on all the arguments */
    int extent;
```
int result = PMPI_Send(buffer,count,datatype,dest,tag,comm);
MPI_Type_size(datatype, &extent); /* Compute size */
totalBytes += count*extent;
totalTime += MPI_Wtime() - tstart; /* and time */
return result;
}

14.4.2 MPI Library Implementation

On a Unix system, in which the MPI library is implemented in C, then there are various possible options, of which two of the most obvious are presented here. Which is better depends on whether the linker and compiler support weak symbols.

Systems with 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 through the use of #pragma weak thus

#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.

Systems Without Weak Symbols
In the absence of weak symbols then one possible solution would be to use the C macro pre-processor thus

#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

#define FUNCTION(name) P##name
#define FUNCTION(name) P/**/name
#define FUNCTION(name) name

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.

Systems Without Weak Symbols
In the absence of weak symbols then one possible solution would be to use the C macro pre-processor thus

#define FUNCTION(name) P##name
#define FUNCTION(name) P/**/name
#define FUNCTION(name) name

Each of the user visible functions in the library would then be declared thus

#define FUNCTION(name) P##name
#define FUNCTION(name) P/**/name
#define FUNCTION(name) name

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 PROFILES 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 she wishes to intercept, 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. libmpi.a contains the “name shifted” MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

14.4.3 Complications

Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g. a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an MPI function that was called from a profiling function. This could lead to “double counting” of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g. it might allow one to answer the question “How much time is spent in the point to point routines when they’re called from collective functions?”), we have decided not to enforce any restrictions on the author of the MPI library that would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it herself. In a single threaded world this is easily achieved through use of a static variable in the profiling code that remembers if you are already inside a profiling routine. It becomes more complex in a multi-threaded environment (as does the meaning of the times recorded!)

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 achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none

The same source file can then be compiled to produce both versions of the library, depending on the state of the PROFILES 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 she wishes to intercept, 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. libmpi.a contains the “name shifted” MPI functions, and libmpi.a contains the normal definitions of the MPI functions.

14.4.3 Complications

Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g. a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an MPI function that was called from a profiling function. This could lead to “double counting” of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g. it might allow one to answer the question “How much time is spent in the point to point routines when they’re called from collective functions?”), we have decided not to enforce any restrictions on the author of the MPI library that would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it herself. In a single threaded world this is easily achieved through use of a static variable in the profiling code that remembers if you are already inside a profiling routine. It becomes more complex in a multi-threaded environment (as does the meaning of the times recorded!)

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 achieved by using wrapper functions on top of the C implementation. The author of the profile library then assumes that it is reasonable only to provide profile functions for the C binding, since Fortran will eventually call these, and the cost of the wrappers is assumed to be small. However, if the wrapper functions are not in the profiling library, then none
14.5 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.
- 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.

Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function.

Unfortunately such an implementation may require more cooperation between the different profiling libraries than is required for the single level implementation detailed above.
Chapter 15

Deprecated Functions

15.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI_TYPE_CREATE_HVECTOR in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

MPI_TYPE_HVECTOR( count, blocklength, stride, oldtype, newtype)

IN count number of blocks (nonnegative integer)
IN blocklength number of elements in each block (nonnegative integer)
IN stride number of bytes between start of each block (integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)

int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE

The following function is deprecated and is superseded by MPI_TYPE_CREATE_HINDEXED in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

The following function is deprecated and is superseded by MPI_TYPE_CREATE_HINDEXED in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.
The following function is deprecated and is superseded by
MPI_TYPE_HINDEXED in MPI-2.0. The language independent definition and the C
binding of the deprecated function is the same as of the new function, except of the function
name. Only the Fortran language binding is different.

MPI_TYPE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, new-
type)

IN  count  number of blocks - also number of entries in
array_of_displacements and array_of_blocklengths (non-
negative integer)
IN  array_of_blocklengths  number of elements in each block (array of nonnega-
tive integers)
IN  array_of_displacements  byte displacement of each block (array of integer)
IN  oldtype  old datatype (handle)
OUT newtype  new datatype (handle)

int MPI_Type_hindexed(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)

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

int MPI_Type_hindexed(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)

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

The following function is deprecated and is superseded by
MPI_TYPE_CREATESTRUCT in MPI-2.0. The language independent definition and the C
binding of the deprecated function is the same as of the new function, except of the function
name. Only the Fortran language binding is different.

MPI_TYPE_CREATESTRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types,
newtype)

IN  count  number of blocks (integer) (nonnegative integer) - also
number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths
IN  array_of_blocklength  number of elements in each block (array of nonnega-
tive integer)
IN  array_of_displacements  byte displacement of each block (array of integer)
IN  array_of_types  type of elements in each block (array of handles to
datatype objects)
OUT newtype  new datatype (handle)

int MPI_Type_struct(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype *newtype)

MPI_TYPE_CREATESTRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
ARRAY_OF_TYPES, NEWTYPE, IERROR)

INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
ARRAY_OF_TYPES(*), NEWTYPE, IERROR
The following function is deprecated and is superseded by MPI\_GET\_ADDRESS in MPI-2.0. The language independent definition and the C binding of the deprecated function is the same as of the new function, except of the function name. Only the Fortran language binding is different.

**MPI\_ADDRESS(location, address)**

<table>
<thead>
<tr>
<th>IN</th>
<th>location</th>
<th>location in caller memory (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>address</td>
<td>address of location (integer)</td>
</tr>
</tbody>
</table>

```c
int MPI\_Address(void* location, MPI\_Aint *address)
```

**MPI\_ADDRESS(LOCATION, ADDRESS, IERROR)**

```c
<stdio> LOCATION(*) INTEGER ADDRESS, IERROR
```

The following functions are deprecated and are superseded by MPI\_TYPE\_GET\_EXTENT in MPI-2.0.

**MPI\_TYPE\_EXTENT(datatype, extent)**

<table>
<thead>
<tr>
<th>IN</th>
<th>datatype</th>
<th>datatype (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>extent</td>
<td>datatype extent (integer)</td>
</tr>
</tbody>
</table>

```c
int MPI\_Type\_extent(MPI\_Datatype datatype, MPI\_Aint *extent)
```

**MPI\_TYPE\_EXTENT(DATATYPE, EXTENT, IERROR)**

```c
<stdio> DATATYPE, EXTENT, IERROR
```

Returns the extent of a datatype, where extent is as defined on page 96. The two functions below can be used for finding the lower bound and the upper bound of a datatype.

**MPI\_TYPE\_LB( datatype, displacement)**

<table>
<thead>
<tr>
<th>IN</th>
<th>datatype</th>
<th>datatype (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>displacement</td>
<td>displacement of lower bound from origin, in bytes (integer)</td>
</tr>
</tbody>
</table>

```c
int MPI\_Type\_lb(MPI\_Datatype datatype, MPI\_Aint* displacement)
```

**MPI\_TYPE\_LB( DATATYPE, DISPLACEMENT, IERROR)**

```c
<stdio> DATATYPE, DISPLACEMENT, IERROR
```

Returns the extent of a datatype, where extent is as defined on page 96. The two functions below can be used for finding the lower bound and the upper bound of a datatype.
MPI_TYPE_UB( datatype, displacement)

IN   datatype   datatype (handle)
OUT  displacement  displacement of upper bound from origin, in bytes (integer)

int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
MPI_TYPE_UB( DATATYPE, DISPLACEMENT, IERROR)

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 of the new function, except of the function name. 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

int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function *
delete_fn, int *keyval, void* extra_state)
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
INTEGER COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR

The copy_fn function is invoked when a communicator is duplicated by
MPI_COMM_CREATE. The language independent definition of the
language bindings are modified.

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:
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,ATTRIBUTE_VAL_OUT, IERR
LOGICAL FLAG

copy_fn may be specified as MPI_NULL_COPY_FN or MPI_DUP_FN from either C or
FORTRAN; MPI_NULL_COPY_FN is a function that does nothing other than returning
flag = 0 and MPI_SUCCESS. MPI_DUP_FN is a simple-minded copy function that sets flag =
1, returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS. Note
that MPI_NULL_COPY_FN and MPI_DUP_FN are also deprecated.
15.1. DEPRECATED SINCE MPI-2.0

Analogous to copy_fn in a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows:

```c
typedef int MPI_Delete_function(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state);
```

A Fortran declaration for such a function is as follows:

```fortran
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR) 
  INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
  integer fn
  delete_fn may be specified as MPI_NULL_DELETE_FN from either C or FORTRAN; 
  MPI_NULL_DELETE_FN is a function that does nothing, other than returning 
  MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.
  The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL 
  in MPI-2.0. The language independent definition of the deprecated function is the same as 
  of the new function, except of the function name. The language bindings are modified.

MPI_KEYVAL_FREE(keyval)
  INOUT keyval Frees the integer key value (integer)

int MPI_Keyval_free(int *keyval)
MPI_KEYVAL_FREE(KEYVAL, IERROR)
  INTEGER KEYVAL, IERROR
  The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in 
  MPI-2.0. The language independent definition of the deprecated function is the same as 
  of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_PUT(comm, keyval, attribute_val)
  INOUT comm communicator to which attribute will be attached (handle)
  keyval key value, as returned by 
  MPI_KEYVAL_CREATE (integer)
  IN attribute_val attribute value

int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
  INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
  The following function is deprecated and is superseded by MPI_COMM_GET_ATTR in 
  MPI-2.0. The language independent definition of the deprecated function is the same as of 
  the new function, except of the function name. The language bindings are modified.

MPI_ATTR_PUT(comm, keyval, attribute_val)
  INOUT comm communicator to which attribute will be attached (handle)
  keyval key value, as returned by 
  MPI_KEYVAL_CREATE (integer)
  IN attribute_val attribute value

int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
  INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
  The following function is deprecated and is superseded by MPI_COMM_GET_ATTR in 
  MPI-2.0. The language independent definition of the deprecated function is the same as of 
  the new function, except of the function name. The language bindings are modified.
```
CHAPTER 15. DEPRECATED FUNCTIONS

**MPI_ATTR_GET(comm, keyval, attribute_val, flag)**

**IN** comm
communicator to which attribute is attached (handle)

**IN** keyval
key value (integer)

**OUT** attribute_val
attribute value, unless flag = false

**OUT** flag
true if an attribute value was extracted; false if no attribute is associated with the key

```c
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
```

The following function is deprecated and is superseded by **MPI_COMM_DELETE_ATTR** in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

**MPI_ATTR_DELETE(comm, keyval)**

**INOUT** comm
communicator to which attribute is attached (handle)

**IN** keyval
The key value of the deleted attribute (integer)

```c
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

The following function is deprecated and is superseded by **MPI_COMM_CREATE_ERRHANDLER** in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

**MPI_ERRHANDLER_CREATE( function, errhandler )**

**IN** function
user defined error handling procedure

**OUT** errhandler
MPI error handler (handle)

```c
int MPI_Errhandler_create(MPI_Handler_function *function, MPI_Errhandler *errhandler)
```

- **MPI_ATTR_GET(comm, keyval, attribute_val, flag)**
- **MPI_ATTR_DELETE(comm, keyval)**
- **MPI_ERRHANDLER_CREATE( function, errhandler )**
In the C language, the user routine should be a C function of type `MPI_Handler_function`, which is defined as:

```c
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use, the second is the error code to be returned.

In the Fortran language, the user routine should be of the form:

```fortran
SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE, .....)  
INTEGER COMM, ERROR_CODE
```

The following function is deprecated and is superseded by `MPI_COMM_SET_ERRHANDLER` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

```c
MPI_ERRHANDLER_SET( comm, errhandler )  
INOUT comm communicator to set the error handler for (handle)  
IN errhandler new MPI error handler for communicator (handle)
```

```c
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
```

The following function is deprecated and is superseded by `MPI_COMM_GET_ERRHANDLER` in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

```c
MPI_ERRHANDLER_GET( comm, errhandler )  
IN comm communicator to get the error handler from (handle)  
OUT errhandler MPI error handler currently associated with communicator (handle)
```

```c
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
```
Chapter 16

Language Bindings

16.1 C++

16.1.1 Overview

There are some issues specific to C++ that must be considered in the design of an interface that go beyond the simple description of language bindings. In particular, in C++, we must be concerned with the design of objects and their interfaces, rather than just the design of a language-specific functional interface to MPI. Fortunately, the design of MPI was based on the notion of objects, so a natural set of classes is already part of MPI.

MPI-2 includes C++ bindings as part of its function specifications. In some cases, MPI-2 provides new names for the C bindings of MPI-1 functions. In this case, the C++ binding matches the new C name — there is no binding for the deprecated name.

16.1.2 Design

The C++ language interface for MPI is designed according to the following criteria:

1. The C++ language interface consists of a small set of classes with a lightweight functional interface to MPI. The classes are based upon the fundamental MPI object types (e.g., communicator, group, etc.).

2. The MPI C++ language bindings provide a semantically correct interface to MPI.

3. To the greatest extent possible, the C++ bindings for MPI functions are member functions of MPI classes.

Rationale. Providing a lightweight set of MPI objects that correspond to the basic MPI types is the best fit to MPI’s implicit object-based design; methods can be supplied for these objects to realize MPI functionality. The existing C bindings can be used in C++ programs, but much of the expressive power of the C++ language is forfeited. On the other hand, while a comprehensive class library would make user programming more elegant, such a library it is not suitable as a language binding for MPI since a binding must provide a direct and unambiguous mapping to the specified functionality of MPI. (End of rationale.)
16.1.3 C++ Classes for MPI

All MPI classes, constants, and functions are declared within the scope of an MPI namespace. Thus, instead of the MPI_ prefix that is used in C and Fortran, MPI functions essentially have an MPI:: prefix.

The members of the MPI namespace are those classes corresponding to objects implicitly used by MPI. An abbreviated definition of the MPI namespace and its member classes is as follows:

```cpp
namespace MPI {
    class Comm {...};
    class Intracom : public Comm {...};
    class Graphcomm : public Intracom {...};
    class Cartcomm : public Intracom {...};
    class Datatype {...};
    class Errhandler {...};
    class Exception {...};
    class File {...};
    class Group {...};
    class Info {...};
    class Op {...};
    class Request : public Request {...};
    class Qrequest : public Request {...};
    class Status {...};
    class Win {...};
};
```

Note that there are a small number of derived classes, and that virtual inheritance is not used.

16.1.4 Class Member Functions for MPI

Besides the member functions which constitute the C++ language bindings for MPI, the C++ language interface has additional functions (as required by the C++ language). In particular, the C++ language interface must provide a constructor and destructor, an assignment operator, and comparison operators.

The complete set of C++ language bindings for MPI is presented in Annex A.4. The bindings take advantage of some important C++ features, such as references and const. Declarations (which apply to all MPI member classes) for construction, destruction, copying, assignment, comparison, and mixed-language operability are also provided.

Except where indicated, all non-static member functions (except for constructors and the assignment operator) of MPI member classes are virtual functions.

**Rationale.** Providing virtual member functions is an important part of design for inheritance. Virtual functions can be bound at run-time, which allows users of libraries to re-define the behavior of objects already contained in a library. There is a small performance penalty that must be paid (the virtual function must be looked up before
it can be called). However, users concerned about this performance penalty can force compile-time function binding. \textit{(End of rationale.)}

Example 16.1

Example showing a derived MPI class.

```cpp
class foo_comm : public MPI::Intracomm { public:
  void Send(const void* buf, int count, const MPI::Datatype& type,
            int dest, int tag) const
  {
    // Class library functionality
    MPI::Intracomm::Send(buf, count, type, dest, tag);
    // More class library functionality
  }
};
```

\textit{Advice to implementors.} Implementors must be careful to avoid unintended side effects from class libraries that use inheritance, especially in layered implementations. For example, if MPI::BCAST is implemented by repeated calls to MPI\_SEND or MPI\_RECV, the behavior of MPI\_BCAST cannot be changed by derived communicator classes that might redefine MPI\_SEND or MPI\_RECV. The implementation of MPI\_BCAST must explicitly use the MPI\_SEND (or MPI\_RECV) of the base MPI::Comm class. \textit{(End of advice to implementors.)}

16.1.5 Semantics

The semantics of the member functions constituting the C++ language binding for MPI are specified by the MPI function description itself. Here, we specify the semantics for those portions of the C++ language interface that are not part of the language binding. In this subsection, functions are prototyped using the type \texttt{MPI::\langle CLASS\rangle} rather than listing each function for every MPI class; the word \texttt{\langle CLASS\rangle} can be replaced with any valid MPI class name (e.g., \texttt{Group}), except as noted.

Construction / Destruction The default constructor and destructor are prototyped as follows:

```cpp
MPI::\langle CLASS\rangle()
~MPI::\langle CLASS\rangle()
```

In terms of construction and destruction, opaque MPI user level objects behave like handles. Default constructors for all MPI objects except MPI::Status create corresponding MPI:\_\*\_NULL handles. That is, when an MPI object is instantiated, comparing it with its corresponding MPI:\_\*\_NULL object will return \textit{true}. The default constructors do not create new MPI opaque objects. Some classes have a member function \texttt{Create()} for this purpose.

Example 16.2

In the following code fragment, the test will return \textit{true} and the message will be sent to \texttt{cout}.

```cpp
Example 16.2
```

\textit{Advice to implementors.} Implementors must be careful to avoid unintended side effects from class libraries that use inheritance, especially in layered implementations. For example, if MPI\_BCAST is implemented by repeated calls to MPI\_SEND or MPI\_RECV, the behavior of MPI\_BCAST cannot be changed by derived communicator classes that might redefine MPI\_SEND or MPI\_RECV. The implementation of MPI\_BCAST must explicitly use the MPI\_SEND (or MPI\_RECV) of the base MPI::Comm class. \textit{(End of advice to implementors.)}

16.1.5 Semantics

The semantics of the member functions constituting the C++ language binding for MPI are specified by the MPI function description itself. Here, we specify the semantics for those portions of the C++ language interface that are not part of the language binding. In this subsection, functions are prototyped using the type \texttt{MPI::\langle CLASS\rangle} rather than listing each function for every MPI class; the word \texttt{\langle CLASS\rangle} can be replaced with any valid MPI class name (e.g., \texttt{Group}), except as noted.

Construction / Destruction The default constructor and destructor are prototyped as follows:

```cpp
MPI::\langle CLASS\rangle()
~MPI::\langle CLASS\rangle()
```

In terms of construction and destruction, opaque MPI user level objects behave like handles. Default constructors for all MPI objects except MPI::Status create corresponding MPI:\_\*\_NULL handles. That is, when an MPI object is instantiated, comparing it with its corresponding MPI:\_\*\_NULL object will return \textit{true}. The default constructors do not create new MPI opaque objects. Some classes have a member function \texttt{Create()} for this purpose.

Example 16.2

In the following code fragment, the test will return \textit{true} and the message will be sent to \texttt{cout}.
void foo()
{
    MPI::Intracomm bar;
    if (bar == MPI::COMM_NULL)
        cout << "bar is MPI::COMM_NULL" << endl;
}

The destructor for each MPI user level object does not invoke the corresponding
MPI_*_FREE function (if it exists).

**Rationale.** MPI_*_FREE functions are not automatically invoked for the following reasons:

1. Automatic destruction contradicts the shallow-copy semantics of the MPI classes.
2. The model put forth in MPI makes memory allocation and deallocation the responsibility of the user, not the implementation.
3. Calling MPI_*_FREE upon destruction could have unintended side effects, including triggering collective operations (this also affects the copy, assignment, and construction semantics). In the following example, we would want neither foo_comm nor bar_comm to automatically invoke MPI_*_FREE upon exit from the function.

```cpp
template
void example_function()
{
    MPI::Intracomm foo_comm(MPI::COMM_WORLD), bar_comm;
    bar_comm = MPI::COMM_WORLD.Dup();
    // rest of function
}
```

(End of rationale.)

**Copy / Assignment** The copy constructor and assignment operator are prototyped as follows:

```cpp
MPI::<CLASS>(const MPI::<CLASS>& data)
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI::<CLASS>& data)
```

In terms of copying and assignment, opaque MPI user level objects behave like handles. Copy constructors perform handle-based (shallow) copies. MPI::Status objects are exceptions to this rule. These objects perform deep copies for assignment and copy construction.

**Advice to implementors.** Each MPI user level object is likely to contain, by value or by reference, implementation-dependent state information. The assignment and copying of MPI object handles may simply copy this value (or reference). (End of advice to implementors.)
Example 16.3 Example using assignment operator. In this example, MPI::Intracomm::Dup() is not called for foo_comm. The object foo_comm is simply an alias for MPI::COMM_WORLD. But bar_comm is created with a call to MPI::Intracomm::Dup() and is therefore a different communicator than foo_comm (and thus different from MPI::COMM_WORLD). baz_comm becomes an alias for bar_comm. If one of bar_comm or baz_comm is freed with MPI_COMM_FREE it will be set to MPI_COMM_NULL. The state of the other handle will be undefined — it will be invalid, but not necessarily set to MPI_COMM_NULL.

```cpp
MPI::Intracomm foo_comm, bar_comm, baz_comm;
foo_comm = MPI::COMM_WORLD;
bar_comm = MPI::COMM_WORLD.Dup();
baz_comm = bar_comm;
```

Comparison The comparison operators are prototyped as follows:

```cpp
bool MPI::<CLASS>::operator==(const MPI::<CLASS>& data) const
bool MPI::<CLASS>::operator!=(const MPI::<CLASS>& data) const
```

The member function operator==() returns true only when the handles reference the same internal MPI object, false otherwise. operator*() returns the boolean complement of operator==(). However, since the Status class is not a handle to an underlying MPI object, it does not make sense to compare Status instances. Therefore, the operator==() and operator*() functions are not defined on the Status class.

Constants Constants are singleton objects and are declared const. Note that not all globally defined MPI objects are constant. For example, MPI::COMM_WORLD and MPI::COMM_SELF are not const.

### 16.1.6 C++ Datatypes

Table 16.1 lists all of the C++ predefined MPI datatypes and their corresponding C and C++ datatypes. Table 16.2 lists all of the Fortran predefined MPI datatypes and their corresponding Fortran 77 datatypes. Table 16.3 lists the C++ names for all other MPI datatypes. MPI::BYTE and MPI::PACKED conform to the same restrictions as MPI::BYTE and MPI::PACKED, listed in Sections 3.2.2 on page 27 and Sections 4.2 on page 120, respectively.

The following table defines groups of MPI predefined datatypes:

| C integer: | MPI::INT, MPI::LONG, MPI::SHORT, MPI::UNSIGNED, MPI::UNSIGNED_LONG, MPI::LONG_LONG, MPI::UNSIGNED_LONG_LONG, MPI::SIGNED_CHAR, MPI::UNSIGNED_CHAR |
| Fortran integer: | MPI::INTEGER |
| Floating point: | MPI::FLOAT, MPI::DOUBLE, MPI::REAL, MPI::DOUBLE_PRECISION, MPI::LONG_DOUBLE |

| C integer: | MPI::INT, MPI::LONG, MPI::SHORT, MPI::UNSIGNED, MPI::UNSIGNED_LONG, MPI::LONG_LONG, MPI::UNSIGNED_LONG_LONG, MPI::SIGNED_CHAR, MPI::UNSIGNED_CHAR |
| Fortran integer: | MPI::INTEGER |
| Floating point: | MPI::FLOAT, MPI::DOUBLE, MPI::REAL, MPI::DOUBLE_PRECISION, MPI::LONG_DOUBLE |
### Table 16.1: C++ names for the MPI C and C++ predefined datatypes, and their corresponding C/C++ datatypes.

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
<th>C++ datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::INT</td>
<td>signed int</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI::LONG</td>
<td>signed long</td>
<td>signed long</td>
</tr>
<tr>
<td>MPI::LONG_LONG</td>
<td>signed long long</td>
<td>signed long long</td>
</tr>
<tr>
<td>MPI::CHAR</td>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>MPI::SHORT</td>
<td>signed short</td>
<td>signed short</td>
</tr>
<tr>
<td>MPI::UNSIGNED_CHAR</td>
<td>unsigned char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI::UNSIGNED_SHORT</td>
<td>unsigned short</td>
<td>unsigned short</td>
</tr>
<tr>
<td>MPI::UNSIGNED</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI::UNSIGNED_LONG</td>
<td>unsigned long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>MPI::DOUBLE_LONG</td>
<td>long double</td>
<td>long double</td>
</tr>
<tr>
<td>MPI::WCHAR</td>
<td>wchar_t</td>
<td>wchar_t</td>
</tr>
<tr>
<td>MPI::BYTE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI::PACKED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 16.2: C++ names for the MPI Fortran predefined datatypes, and their corresponding Fortran 77 datatypes.

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>Fortran datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI::REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI::DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI::COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI::LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI::CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI::BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI::PACKED</td>
<td></td>
</tr>
</tbody>
</table>

### Table 16.3: C++ names for the MPI Fortran predefined datatypes, and their corresponding Fortran 77 datatypes.

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>Fortran datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI::REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI::DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI::COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI::LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI::CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI::BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI::PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Table 16.3: C++ names for other MPI datatypes. Implementations may also define other optional types (e.g., MPI::INTEGER8).

Logical:
- MPI::LOGICAL, MPI::BOOL

Complex:
- MPI::F_COMPLEX, MPI::COMPLEX,
- MPI::DOUBLE_COMPLEX,
- MPI::LONG_DOUBLE_COMPLEX

Byte:
- MPI::BYTE

Valid datatypes for each reduction operation are specified below in terms of the groups defined above.

<table>
<thead>
<tr>
<th>Op</th>
<th>Allowed Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::MAX, MPI::MIN</td>
<td>C integer, Fortran integer, Floating point</td>
</tr>
<tr>
<td>MPI::SUM, MPI::PROD</td>
<td>C integer, Fortran integer, Floating point, Complex</td>
</tr>
<tr>
<td>MPI::LAND, MPI::LOR, MPI::LXOR</td>
<td>C integer, Logical</td>
</tr>
<tr>
<td>MPI::BAND, MPI::BOR, MPI::BXOR</td>
<td>C integer, Fortran integer, Byte</td>
</tr>
</tbody>
</table>

MPI::MINLOC and MPI::MAXLOC perform just as their C and Fortran counterparts; see Section 5.9.4 on page 164.

16.1.7 Communicators

The MPI::Comm class hierarchy makes explicit the different kinds of communicators implicitly defined by MPI and allows them to be strongly typed. Since the original design of MPI defined only one type of handle for all types of communicators, the following clarifications are provided for the C++ design.
Types of communicators

There are five different types of communicators: MPI::Comm, MPI::Intercomm, MPI::Intracomm, MPI::Cartcomm, and MPI::Graphcomm. MPI::Comm is the abstract base communicator class, encapsulating the functionality common to all MPI communicators. MPI::Intercomm and MPI::Intracomm are derived from MPI::Comm. MPI::Cartcomm and MPI::Graphcomm are derived from MPI::Intercomm.

Advice to users. Initializing a derived class with an instance of a base class is not legal in C++. For instance, it is not legal to initialize a Cartcomm from an Intracomm. Moreover, because MPI::Comm is an abstract base class, it is non-instantiable, so that it is not possible to have an object of class MPI::Comm. However, it is possible to have a reference or a pointer to an MPI::Comm.

Example 16.4 The following code is erroneous.

```
Intracomm intra = MPI::COMM_WORLD.Dup();
Cartcomm cart(intra);  // This is erroneous
```

(End of advice to users.)

MPI::COMM_NULL. The specific type of MPI::COMM_NULL is implementation dependent. MPI::COMM_NULL must be able to be used in comparisons and initializations with all types of communicators. MPI::COMM_NULL must also be able to be passed to a function that expects a communicator argument in the parameter list (provided that MPI::COMM_NULL is an allowed value for the communicator argument).

Rationale. There are several possibilities for implementation of MPI::COMM_NULL. Specifying its required behavior, rather than its realization, provides maximum flexibility to implementors. (End of rationale.)

Example 16.5 The following example demonstrates the behavior of assignment and comparison using MPI::COMM_NULL.

```
MPI::Intercomm comm;
comm = MPI::COMM_NULL;  // assign with COMM_NULL
if (comm == MPI::COMM_NULL)  // true
  cout << "comm is NULL" << endl;
if (MPI::COMM_NULL == comm)  // note -- a different function!
  cout << "comm is still NULL" << endl;
```

```
Dup() is not defined as a member function of MPI::Comm, but it is defined for the derived classes of MPI::Comm. Dup() is not virtual and it returns its OUT parameter by value.
```

MPI::COMM_NULL. The specific type of MPI::COMM_NULL is implementation dependent. MPI::COMM_NULL must be able to be used in comparisons and initializations with all types of communicators. MPI::COMM_NULL must also be able to be passed to a function that expects a communicator argument in the parameter list (provided that MPI::COMM_NULL is an allowed value for the communicator argument).

Rationale. There are several possibilities for implementation of MPI::COMM_NULL. Specifying its required behavior, rather than its realization, provides maximum flexibility to implementors. (End of rationale.)

Example 16.5 The following example demonstrates the behavior of assignment and comparison using MPI::COMM_NULL.

```
MPI::Intercomm comm;
comm = MPI::COMM_NULL;  // assign with COMM_NULL
if (comm == MPI::COMM_NULL)  // true
  cout << "comm is NULL" << endl;
if (MPI::COMM_NULL == comm)  // note -- a different function!
  cout << "comm is still NULL" << endl;
```

```
Dup() is not defined as a member function of MPI::Comm, but it is defined for the derived classes of MPI::Comm. Dup() is not virtual and it returns its OUT parameter by value.
```
MPI::Comm::Clone() The C++ language interface for MPI includes a new function Clone(). MPI::Comm::Clone() is a pure virtual function. For the derived communicator classes, Clone() behaves like Dup() except that it returns a new object by reference. The Clone() functions are prototyped as follows:

```cpp
Comm& Comm::Clone() const = 0
Intracomm& Intracomm::Clone() const
Intercomm& Intercomm::Clone() const
Cartcomm& Cartcomm::Clone() const
Graphcomm& Graphcomm::Clone() const
```

Rationale. Clone() provides the “virtual dup” functionality that is expected by C++ programmers and library writers. Since Clone() returns a new object by reference, users are responsible for eventually deleting the object. A new name is introduced rather than changing the functionality of Dup(). (End of rationale.)

Advice to implementors. Within their class declarations, prototypes for Clone() and Dup() would look like the following:

```cpp
namespace MPI {
    class Comm {
        virtual Comm& Clone() const = 0;
    };
    class Intracomm : public Comm {
        Intracomm Dup() const { ... }
        virtual Intracomm& Clone() const { ... };
    };
    class Intercomm : public Comm {
        Intercomm Dup() const { ... }
        virtual Intercomm& Clone() const { ... };
    };
    // Cartcomm and Graphcomm are similarly defined
};
```

(End of advice to implementors.)

16.1.8 Exceptions
The C++ language interface for MPI includes the predefined error handler

```cpp
MPI::ERRORS_THROW_EXCEPTIONS for use with the Set_errhandler() member functions.
MPI::ERRORS_THROW_EXCEPTIONS can only be set or retrieved by C++ functions. If a non-C++ program causes an error that invokes the MPI::ERRORS_THROW_EXCEPTIONS error handler, the exception will pass up the calling stack until C++ code can catch it. If there is no C++ code to catch it, the behavior is undefined. In a multi-threaded environment or if a non-blocking MPI call throws an exception while making progress in the background, the behavior is implementation dependent.
```
The error handler MPI::ERRORS_THROW_EXCEPTIONS causes an MPI::Exception to be thrown for any MPI result code other than MPI::SUCCESS. The public interface to MPI::Exception class is defined as follows:

```cpp
namespace MPI {
    class Exception {
        public:
            Exception(int error_code);
            int Get_error_code() const;
            int Get_error_class() const;
            const char *Get_error_string() const;
    };
}
```

**Advice to implementors.**

The exception will be thrown within the body of MPI::ERRORS_THROW_EXCEPTIONS. It is expected that control will be returned to the user when the exception is thrown. Some MPI functions specify certain return information in their parameters in the case of an error and MPI_ERRORS_RETURN is specified. The same type of return information must be provided when exceptions are thrown.

For example, MPI_WAITALL puts an error code for each request in the corresponding entry in the status array and returns MPI_ERR_IN_STATUS. When using MPI::ERRORS_THROW_EXCEPTIONS, it is expected that the error codes in the status array will be set appropriately before the exception is thrown.

(End of advice to implementors.)

### 16.1.9 Mixed-Language Operability

The C++ language interface provides functions listed below for mixed-language operability. These functions provide for a seamless transition between C and C++. For the case where the C++ class corresponding to `<CLASS>` has derived classes, functions are also provided for converting between the derived classes and the C MPI_<CLASS>.

```cpp
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI_<CLASS>& data)
MPI::<CLASS>(const MPI_<CLASS>& data)
MPI::<CLASS>::operator MPI_<CLASS>() const
```

These functions are discussed in Section 16.3.4.

### 16.1.10 Profiling

This section specifies the requirements of a C++ profiling interface to MPI.

**Advice to implementors.** Since the main goal of profiling is to intercept function calls from user code, it is the implementor’s decision how to layer the underlying implementation to allow function calls to be intercepted and profiled. If an implementation

```cpp
MPI::ERRORS_THROW_EXCEPTIONS
```
of the MPI C++ bindings is layered on top of MPI bindings in another language (such as C), or if the C++ bindings are layered on top of a profiling interface in another language, no extra profiling interface is necessary because the underlying MPI implementation already meets the MPI profiling interface requirements.

Native C++ MPI implementations that do not have access to other profiling interfaces must implement an interface that meets the requirements outlined in this section.

High-quality implementations can implement the interface outlined in this section in order to promote portable C++ profiling libraries. Implementors may wish to provide an option whether to build the C++ profiling interface or not; C++ implementations that are already layered on top of bindings in another language or another profiling interface will have to insert a third layer to implement the C++ profiling interface.

(End of advice to implementors.)

To meet the requirements of the C++ MPI profiling interface, an implementation of the MPI functions must:

1. Provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix “MPI::”') should also be accessible with the prefix “PMPI::”.

2. Ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.

3. Document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that profiler developer knows whether they must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.

4. Where the implementation of different language bindings is done through a layered approach (e.g., the C++ binding is a set of “wrapper” functions which call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

This 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 author of 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.

5. Provide a no-op routine MPI::Pcontrol in the MPI library.

Advice to implementors. There are (at least) two apparent options for implementing the C++ profiling interface: inheritance or caching. An inheritance-based approach may not be attractive because it may require a virtual inheritance implementation of the communicator classes. Thus, it is most likely that implementors will cache PMPI objects on their corresponding MPI objects. The caching scheme is outlined below.

The “real” entry points to each routine can be provided within a namespace PMPI. The non-profiling version can then be provided within a namespace MPI.
Caching instances of PMPI objects in the MPI handles provides the “has a” relationship that is necessary to implement the profiling scheme.

Each instance of an MPI object simply “wraps up” an instance of a PMPI object. MPI objects can then perform profiling actions before invoking the corresponding function in their internal PMPI object.

The key to making the profiling work by simply re-linking programs is by having a header file that declares all the MPI functions. The functions must be defined elsewhere, and compiled into a library. MPI constants should be declared extern in the MPI namespace. For example, the following is an excerpt from a sample mpi.h file:

```c
namespace PMPI {
    class Comm {
        public:
            int Get_size() const;
        };// etc.
    };

    namespace MPI {
        public:
            class Comm {
                public:
                    int Get_size() const;
                private:
                    PMPI::Comm pmpi_comm;
                };
    };
}
```

Note that all constructors, the assignment operator, and the destructor in the MPI class will need to initialize/destroy the internal PMPI object as appropriate.

The definitions of the functions must be in separate object files; the PMPI class member functions and the non-profiling versions of the MPI class member functions can be compiled into `libmpi.a`, while the profiling versions can be compiled into `libpmpi.a`. Note that the PMPI class member functions and the MPI constants must be in different object files than the non-profiling MPI class member functions in the `libmpi.a` library to prevent multiple definitions of MPI class member function names when linking both `libmpi.a` and `libpmpi.a`. For example:

```c
Example 16.7 pmpi.cc, to be compiled into libmpi.a.
int PMPI::Comm::Get_size() const
{
    // Implementation of MPI_COMM_SIZE
}
```

Caching instances of PMPI objects in the MPI handles provides the “has a” relationship that is necessary to implement the profiling scheme.

Each instance of an MPI object simply “wraps up” an instance of a PMPI object. MPI objects can then perform profiling actions before invoking the corresponding function in their internal PMPI object.

The key to making the profiling work by simply re-linking programs is by having a header file that declares all the MPI functions. The functions must be defined elsewhere, and compiled into a library. MPI constants should be declared extern in the MPI namespace. For example, the following is an excerpt from a sample mpi.h file:

```c
namespace PMPI {
    class Comm {
        public:
            int Get_size() const;
        };// etc.
    };

    namespace MPI {
        public:
            class Comm {
                public:
                    int Get_size() const;
                private:
                    PMPI::Comm pmpi_comm;
                };
    };
}
```

Note that all constructors, the assignment operator, and the destructor in the MPI class will need to initialize/destroy the internal PMPI object as appropriate.

The definitions of the functions must be in separate object files; the PMPI class member functions and the non-profiling versions of the MPI class member functions can be compiled into `libmpi.a`, while the profiling versions can be compiled into `libpmpi.a`. Note that the PMPI class member functions and the MPI constants must be in different object files than the non-profiling MPI class member functions in the `libmpi.a` library to prevent multiple definitions of MPI class member function names when linking both `libmpi.a` and `libpmpi.a`. For example:

```c
Example 16.7 pmpi.cc, to be compiled into libmpi.a.
int PMPI::Comm::Get_size() const
{
    // Implementation of MPI_COMM_SIZE
}
```
16.2. FORTRAN SUPPORT

Example 16.8 constants.cc, to be compiled into libmpi.a.
const MPI::Intracomm MPI::COMM_WORLD;

Example 16.9 mpi_no_profile.cc, to be compiled into libmpi.a.
int MPI::Comm::Get_size() const
{
    return pmpi_comm.Get_size();
}

Example 16.10 mpi_profile.cc, to be compiled into libpmpi.a.
int MPI::Comm::Get_size() const
{
    // Do profiling stuff
    int ret = pmpi_comm.Get_size();
    // More profiling stuff
    return ret;
}

(End of advice to implementors.)

16.2 Fortran Support

16.2.1 Overview

Fortran 90 is the current international Fortran standard. MPI-2 Fortran bindings are Fortran 90 bindings that in most cases are “Fortran 77 friendly.” That is, with few exceptions (e.g., KIND-parameterized types, and the mpi module, both of which can be avoided) Fortran 77 compilers should be able to compile MPI programs.

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. MPI does not (yet) use many of these features because of a number of technical difficulties.

(End of rationale.)

MPI defines two levels of Fortran support, described in Sections 16.2.3 and 16.2.4. A third level of Fortran support is envisioned, but is deferred to future standardization efforts. In the rest of this section, “Fortran” shall refer to Fortran 90 (or its successor) unless qualified.

1. Basic Fortran Support An implementation with this level of Fortran support provides the original Fortran bindings specified in MPI-1, with small additional requirements specified in Section 16.2.3.

Example 16.8 constants.cc, to be compiled into libmpi.a.
const MPI::Intracomm MPI::COMM_WORLD;

Example 16.9 mpi_no_profile.cc, to be compiled into libmpi.a.
int MPI::Comm::Get_size() const
{
    return pmpi_comm.Get_size();
}

Example 16.10 mpi_profile.cc, to be compiled into libpmpi.a.
int MPI::Comm::Get_size() const
{
    // Do profiling stuff
    int ret = pmpi_comm.Get_size();
    // More profiling stuff
    return ret;
}

(End of advice to implementors.)

16.2 Fortran Support

16.2.1 Overview

Fortran 90 is the current international Fortran standard. MPI-2 Fortran bindings are Fortran 90 bindings that in most cases are “Fortran 77 friendly.” That is, with few exceptions (e.g., KIND-parameterized types, and the mpi module, both of which can be avoided) Fortran 77 compilers should be able to compile MPI programs.

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. MPI does not (yet) use many of these features because of a number of technical difficulties.

(End of rationale.)

MPI defines two levels of Fortran support, described in Sections 16.2.3 and 16.2.4. A third level of Fortran support is envisioned, but is deferred to future standardization efforts. In the rest of this section, “Fortran” shall refer to Fortran 90 (or its successor) unless qualified.

1. Basic Fortran Support An implementation with this level of Fortran support provides the original Fortran bindings specified in MPI-1, with small additional requirements specified in Section 16.2.3.
2. Extended Fortran Support
An implementation with this level of Fortran support provides Basic Fortran Support plus additional features that specifically support Fortran 90, as described in Section 16.2.4.

A compliant MPI-2 implementation providing a Fortran interface must provide Extended Fortran Support unless the target compiler does not support modules or KIND-parameterized types.

16.2.2 Problems With Fortran Bindings for MPI
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 does not add to the standard, but is intended to clarify the standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these 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. 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. It supersedes and replaces the discussion of Fortran bindings in the original MPI specification (for Fortran 90, not Fortran 77).

The following MPI features are inconsistent with Fortran 90:

1. An MPI subroutine with a choice argument may be called with different argument types.
2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.
3. Many MPI routines assume that actual arguments are passed by address and that arguments are not copied on entrance to or exit from the subroutine.
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.
5. Several named “constants,” such as MPI_BOTTOM, MPI_INPLACE, MPI_STATUS_IGNORE, MPISTATUSES_IGNORE, MPIERRCODES_IGNORE, MPI_ARGV_NULL, and MPIARGV_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 14 for more information.
6. The memory allocation routine MPI_ALLOC_MEM can’t be usefully used in Fortran without a language extension that allows the allocated memory to be associated with a Fortran variable.

MPI-1 contained several routines that take address-sized information as input or return 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 KIND=MPI_ADDRESS_KIND. A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6 on page 15 and Section 4.1.1 on page 79 for more information.

2. Extended Fortran Support
An implementation with this level of Fortran support provides Basic Fortran Support plus additional features that specifically support Fortran 90, as described in Section 16.2.4.

A compliant MPI-2 implementation providing a Fortran interface must provide Extended Fortran Support unless the target compiler does not support modules or KIND-parameterized types.

16.2.2 Problems With Fortran Bindings for MPI
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 does not add to the standard, but is intended to clarify the standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these 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. 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. It supersedes and replaces the discussion of Fortran bindings in the original MPI specification (for Fortran 90, not Fortran 77).

The following MPI features are inconsistent with Fortran 90:

1. An MPI subroutine with a choice argument may be called with different argument types.
2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.
3. Many MPI routines assume that actual arguments are passed by address and that arguments are not copied on entrance to or exit from the subroutine.
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.
5. Several named “constants,” such as MPI_BOTTOM, MPI_INPLACE, MPI_STATUS_IGNORE, MPISTATUSES_IGNORE, MPIERRCODES_IGNORE, MPI_ARGV_NULL, and MPIARGV_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 14 for more information.
6. The memory allocation routine MPI_ALLOC_MEM can’t be usefully used in Fortran without a language extension that allows the allocated memory to be associated with a Fortran variable.

MPI-1 contained several routines that take address-sized information as input or return 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 KIND=MPI_ADDRESS_KIND. A number of new MPI-2 functions also take INTEGER arguments of non-default KIND. See Section 2.6 on page 15 and Section 4.1.1 on page 79 for more information.
16.2. FORTRAN SUPPORT

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 is technically only allowed if the function is overloaded with a different function for each type. In C, the use of void* formal arguments avoids these problems.

The following code fragment is technically illegal and may generate a compile-time error.

```fortran
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, though there is concern that Fortran 90 compilers are more likely to return errors. It is also technically illegal in Fortran to pass a scalar actual argument to an array dummy argument. Thus the following code fragment may generate an error since the buf argument to MPI_SEND is declared as an assumed-size array <type> buf(*)).

```fortran
integer a
call mpi_send(a, 1, MPI_INTEGER, ...)  
```

Advice to users. In the event that you run into one of the problems related to type checking, you may be able to work around it by using a compiler flag, by compiling separately, or by using an MPI implementation with Extended Fortran Support as described in Section 16.2.4. An alternative that will usually work with variables local to a routine but not with arguments to a function or subroutine is to use the EQUIVALENCE statement to create another variable with a type accepted by the compiler. (End of advice to users.)

Problems Due to Data Copying and Sequence Association

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 90, user data is not necessarily stored contiguously. For example, the array section A(:,1:2) 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(D)) or is of assumed size (e.g., B(*)). If necessary, they do this by making a copy of the array into contiguous memory. Both Fortran 77 and Fortran 90 are carefully worded to allow such copying to occur, but few Fortran 77 compilers do it.

Because MPI dummy buffer arguments are assumed-size arrays, this leads to a serious problem for a non-blocking 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:

```
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, though there is concern that Fortran 90 compilers are more likely to return errors. It is also technically illegal in Fortran to pass a scalar actual argument to an array dummy argument. Thus the following code fragment may generate an error since the buf argument to MPI_SEND is declared as an assumed-size array <type> buf(*)).

```
integer a
call mpi_send(a, 1, MPI_INTEGER, ...)  
```

Advice to users. In the event that you run into one of the problems related to type checking, you may be able to work around it by using a compiler flag, by compiling separately, or by using an MPI implementation with Extended Fortran Support as described in Section 16.2.4. An alternative that will usually work with variables local to a routine but not with arguments to a function or subroutine is to use the EQUIVALENCE statement to create another variable with a type accepted by the compiler. (End of advice to users.)

Problems Due to Data Copying and Sequence Association

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 90, user data is not necessarily stored contiguously. For example, the array section A(:,1:2) 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(D)) or is of assumed size (e.g., B(*)). If necessary, they do this by making a copy of the array into contiguous memory. Both Fortran 77 and Fortran 90 are carefully worded to allow such copying to occur, but few Fortran 77 compilers do it.

Because MPI dummy buffer arguments are assumed-size arrays, this leads to a serious problem for a non-blocking 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:

```
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, though there is concern that Fortran 90 compilers are more likely to return errors. It is also technically illegal in Fortran to pass a scalar actual argument to an array dummy argument. Thus the following code fragment may generate an error since the buf argument to MPI_SEND is declared as an assumed-size array <type> buf(*)).

```
integer a
call mpi_send(a, 1, MPI_INTEGER, ...)  
```

Advice to users. In the event that you run into one of the problems related to type checking, you may be able to work around it by using a compiler flag, by compiling separately, or by using an MPI implementation with Extended Fortran Support as described in Section 16.2.4. An alternative that will usually work with variables local to a routine but not with arguments to a function or subroutine is to use the EQUIVALENCE statement to create another variable with a type accepted by the compiler. (End of advice to users.)

1Technically, the Fortran standards are worded to allow non-contiguous storage of any array data.
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. Sometimes later, MPI may write to the address of
the deallocated temporary. Copying is also a problem for MPI_IDSEND 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 'simple' section such as A(1:3) of such an
array. (We define 'simple' more fully 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 discontinuous, copying is often done. It is this copying that causes
problems for MPI as described in the previous paragraph.

Our formal definition of a 'simple' array section is

name ( [:]... [<subscript>]:[<subscript>] [,<subscript>]... )

That is, there are zero or more dimensions that are selected in full, then one dimension
selected without a stride, then zero or more dimensions that are selected with a simple
subscript. Examples are

A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)

Because of Fortran's column-major ordering, where the first index varies fastest, a simple
section of a contiguous array will also be contiguous.\(^2\)

The same problem can occur with a scalar argument. Some compilers, even for Fortran
77, make a copy of some scalar dummy arguments within a called procedure. That this can
cause a problem is illustrated by the example

call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a

subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end

If a is copied, MPI_IRECV will alter the copy when it completes the communication
and will not alter a itself.

Note that copying will almost certainly occur for an argument that is a non-trivial
expression (one with at least one operator or function call), a section that does not select a
contiguous part of its parent (e.g., A(1:n:2)), a pointer whose target is such a section, or
an assumed-shape array that is (directly or indirectly) associated with such a section.

\(^2\)To keep the definition of 'simple' simple, we have chosen to require all but one of the section subscripts
to be without bounds. A colon without bounds makes it obvious both to the compiler and to the reader
that the whole of the dimension is selected. It would have been possible to allow cases where the whole
dimension is selected with one or two bounds, but this means for the reader that the array declaration or
most recent allocation has to be consulted and for the compiler that a run-time check may be required.

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. Sometimes later, MPI may write to the address of
the deallocated temporary. Copying is also a problem for MPI_IDSEND 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 'simple' section such as A(1:3) of such an
array. (We define 'simple' more fully 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 discontinuous, copying is often done. It is this copying that causes
problems for MPI as described in the previous paragraph.

Our formal definition of a 'simple' array section is

name ( [:]... [<subscript>]:[<subscript>] [,<subscript>]... )

That is, there are zero or more dimensions that are selected in full, then one dimension
selected without a stride, then zero or more dimensions that are selected with a simple
subscript. Examples are

A(1:N), A(:,N), A(:,1:N,1), A(1:6,N), A(:,:,1:N)

Because of Fortran's column-major ordering, where the first index varies fastest, a simple
section of a contiguous array will also be contiguous.\(^2\)

The same problem can occur with a scalar argument. Some compilers, even for Fortran
77, make a copy of some scalar dummy arguments within a called procedure. That this can
cause a problem is illustrated by the example

call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a

subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end

If a is copied, MPI_IRECV will alter the copy when it completes the communication
and will not alter a itself.

Note that copying will almost certainly occur for an argument that is a non-trivial
expression (one with at least one operator or function call), a section that does not select a
contiguous part of its parent (e.g., A(1:n:2)), a pointer whose target is such a section, or
an assumed-shape array that is (directly or indirectly) associated with such a section.

\(^2\)To keep the definition of 'simple' simple, we have chosen to require all but one of the section subscripts
to be without bounds. A colon without bounds makes it obvious both to the compiler and to the reader
that the whole of the dimension is selected. It would have been possible to allow cases where the whole
dimension is selected with one or two bounds, but this means for the reader that the array declaration or
most recent allocation has to be consulted and for the compiler that a run-time check may be required.
If there is a compiler option that inhibits copying of arguments, in either the calling or called procedure, this should be employed.

If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, simple array sections of such arrays, or scalars, and if there is no compiler option to inhibit this, then the compiler cannot be used for applications that use MPI_GET_ADDRESS, or any non-blocking 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 that use memory references across subroutine calls as in the example above.

Special Constants

MPI requires a number of special “constants” that cannot be implemented as normal Fortran constants, including MPI_BOTTOM, MPI_STATUS_IGNORE, MPI_IN_PLACE, MPI_STATUSES_IGNORE and MPI_ERRCODES_IGNORE. 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 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 legal 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).

Fortran 90 Derived Types

MPI does not explicitly support passing Fortran 90 derived types to choice dummy arguments. Indeed, for MPI implementations that provide explicit interfaces through the mpi module a compiler will reject derived type actual arguments at compile time. Even when no explicit interfaces are given, users should be aware that Fortran 90 provides no guarantee of sequence association for derived types or arrays of derived types. For instance, an array of a derived type consisting of two elements may be implemented as an array of the first elements followed by an array of the second. Use of the SEQUENCE attribute may help here, somewhat.

The following code fragment shows one possible way to send a derived type in Fortran.

```fortran
  type mytype
    integer i
    real x
  end type mytype

  type(mytype) foo
  integer blocklen(3), type(3)
  integer(MPI_ADDRESS_KIND) disp(3), base
end type mytype
```

The example assumes that all data is passed by address.
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)

base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
disp(3) = disp(3) - base

blocklen(1) = 1
blocklen(2) = 1
blocklen(3) = 1

type(1) = MPI_INTEGER
type(2) = MPI_REAL
type(3) = MPI_DOUBLE_PRECISION
call MPI_TYPE_CREATE_STRUCT(3, blocklen, disp, type, newtype, ierr)
call MPI_TYPE_COMMIT(newtype, ierr)

! unpleasant to send foo%i instead of foo, but it works for scalar
! entities of type mytype
! call MPI_SEND(foo%i, 1, newtype, ...)
Example 16.11 Fortran 90 register optimization.

This source can be compiled as:

call MPI_GET_ADDRESS(buf,bufaddr, ierror)
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,
MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf
val_new = buf
call MPI_RECV(MPI_BOTTOM,1,type,...)
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,
MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf
val_new = buf
call MPI_RECV(MPI_BOTTOM,...)
call MPI_Type_commit(type)

The compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access of buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

Example 16.12 shows extreme, but allowed, possibilities.

Example 16.12 Fortran 90 register optimization – extreme.

Source compiled as or compiled as

call MPI_RECV(buf,...req) call MPI_RECV(buf,...req) call MPI_RECV(buf,...req)
register = buf 
val_old = register 
call MPI_RECV(MPI_BOTTOM,1,type,...) call MPI_RECV(MPI_BOTTOM,...)
val_new = buf register = buf 
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,MPI_REAL,type,ierror)
call MPI_GET_ADDRESS(buf,bufaddr, ierror)
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,
MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf
val_new = buf
call MPI_RECV(MPI_BOTTOM,1,type,...)
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,MPI_REAL,type,ierror)
call MPI_GET_ADDRESS(buf,bufaddr, ierror)
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,
MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf
val_new = buf
call MPI_RECV(MPI_BOTTOM,...)
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,MPI_REAL,type,ierror)
call MPI_GET_ADDRESS(buf,bufaddr, ierror)
call MPI_TYPE_CREATE_STRUCT(1,1, bufaddr,
MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf
val_new = buf
call MPI_RECV(MPI_BOTTOM,...)

MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_RECV and the finish of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_RECV has returned, and may schedule the load of buf earlier than typed in the source. It has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as in the case on the right.

To prevent instruction reordering or the allocation of a buffer in a register there are two possibilities in portable Fortran code:

- The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. Note that if the intent is declared in 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, the above call of MPI_RECV might be replaced by

    call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)

- The compiler may be prevented from moving a reference to a buffer across a call to an MPI subroutine by surrounding the call by calls to an external subroutine with the buffer as an actual argument. Note that if the intent is declared in 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, the above call of MPI_RECV might be replaced by

    call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
with the separately compiled

```
subroutine DD(buf)
  integer buf
  end
```

(assuming that `buf` has type `INTEGER`). The compiler may be similarly prevented from moving a reference to a variable across a call to an MPI subroutine.

In the case of a non-blocking call, as in the above call of `MPI_WAIT`, no reference to the buffer is permitted until it has been verified that the transfer has been completed.

Therefore, in this case, the extra call ahead of the MPI call is not necessary, i.e., the call of `MPI_WAIT` in the example might be replaced by

```
call MPI_WAIT(req,...)
call DD(buf)
```

• An alternative is to put the buffer or variable into a module or a common block and access it through a `USE` or `COMMON` statement in each scope where it is referenced, defined or appears as an actual argument in a call to an MPI routine. The compiler will then have to assume that the MPI procedure (`MPI_RECV` in the above example) may alter the buffer or variable, provided that the compiler cannot analyze that the MPI procedure does not reference the module or common block.

In the longer term, the attribute `VOLATILE` is under consideration for Fortran 2000 and would give the buffer or variable the properties needed, but it would inhibit optimization of any code containing the buffer or variable.

In C, subroutines which modify variables that are not in the argument list will not cause register optimization problems. This is because taking pointers to storage objects by using the `&` operator and later referencing the objects by way of the pointer is an integral part of the language. A C compiler understands the implications, so that the problem should not occur, in general. However, some compilers do offer optional aggressive optimization levels which may not be safe.

16.2.3 Basic Fortran Support

Because Fortran 90 is (for all practical purposes) a superset of Fortran 77, Fortran 90 (and future) programs can use the original Fortran interface. The following additional requirements are added:

1. Implementations are required to provide the file `mpif.h`, as described in the original MPI-1 specification.

2. `mpif.h` must be valid and equivalent for both fixed- and free-source form.

   **Advice to implementors.** To make `mpif.h` compatible with both fixed- and free-source forms, to allow automatic inclusion by preprocessors, and to allow extended fixed-form line length, it is recommended that requirement two be met by constructing `mpif.h` without any continuation lines. This should be possible because `mpif.h` contains only declarations, and because common block declarations can be split among several lines. To support Fortran 77 as well as Fortran 90, it may be necessary to eliminate all comments from `mpif.h`. (End of advice to implementors.)
16.2.4 Extended Fortran Support

Implementations with Extended Fortran support must provide:

1. An mpi module

2. A new set of functions to provide additional support for Fortran intrinsic numeric types, including parameterized types: MPI_SIZEOF, MPI_TYPE_MATCH_SIZE, MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL and MPI_TYPE_CREATE_F90_COMPLEX. Parameterized types are Fortran intrinsic types which are specified using KIND type parameters. These routines are described in detail in Section 16.2.5.

Additionally, high-quality implementations should provide a mechanism to prevent fatal type mismatch errors for MPI routines with choice arguments.

The mpi Module

An MPI implementation must provide a module named mpi that can be used in a Fortran 90 program. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.

An MPI implementation may provide in the mpi module other features that enhance the usability of MPI while maintaining adherence to the standard. For example, it may:

- Provide interfaces for all or for a subset of MPI routines.
- Provide INTENT information in these interface blocks.

Advice to implementors. The appropriate INTENT may be different from what is given in the MPI generic interface. Implementations must choose INTENT so that the function adheres to the MPI standard. (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 as 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 is changed in several places by MPI-2. For instance, MPI_IN_PLACE changes the sense of an OUT argument to be INOUT. (End of rationale.)

Applications may use either the mpi module or the mpif.h include file. An implementation may require use of the module to prevent type mismatch errors (see below).

Advice to users. It is recommended to use the mpi module even if it is not necessary to use it to avoid type mismatch errors on a particular system. Using a module provides several potential advantages over using an include file. (End of advice to users.)

It must be possible to link together routines some of which use mpi and others of which INCLUDE mpif.h.
Chapter 16. Language Bindings

No Type Mismatch Problems for Subroutines with Choice Arguments

A high-quality MPI implementation should provide a mechanism to ensure that MPI choice arguments do not cause fatal compile-time or run-time errors due to type mismatch. An MPI implementation may require applications to use the mpi module, or require that it be compiled with a particular compiler flag, in order to avoid type mismatch problems.

Advice to implementors. In the case where the compiler does not generate errors, nothing needs to be done to the existing interface. In the case where the compiler may generate errors, a set of overloaded functions may be used. See the paper of M. Hennecke [26]. Even if the compiler does not generate errors, explicit interfaces for all routines would be useful for detecting errors in the argument list. Also, explicit interfaces which give INTENT information can reduce the amount of copying for BUF(*) arguments. (End of advice to implementors.)

16.2.5 Additional Support for Fortran Numeric Intrinsic Types

The routines in this section are part of Extended Fortran Support described in Section 16.2.4.

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 types are declared using an intrinsic type (one of INTEGER, REAL, COMPLEX, LOGICAL and CHARACTER) with an optional integer KIND parameter that selects from among one or more variants. The specific meaning of different KIND values themselves are implementation dependent and not specified by the language. Fortran provides the KIND selection functions selected_real_kind for REAL and COMPLEX types, and selected_int_kind for INTEGER types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare KIND-parameterized REAL, COMPLEX and INTEGER variables in Fortran. This scheme is backward compatible with Fortran 77. REAL and INTEGER Fortran variables have a default KIND if none is specified. Fortran DOUBLE PRECISION variables are of intrinsic type REAL with a non-default KIND. The following two declarations are equivalent:

```fortran
    double precision x
    real(KIND(0.0d0)) x
```

MPI provides two orthogonal methods to communicate using numeric intrinsic types. The first method can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion in heterogeneous environments. The second method gives the user complete control over communication by exposing machine representations.

```fortran
    double precision x
    real(KIND(0.0d0)) x
```

MPI provides two orthogonal methods to communicate using numeric intrinsic types. The first method can be used when variables have been declared in a portable way — using default KIND or using KIND parameters obtained with the selected_int_kind or selected_real_kind functions. With this method, MPI automatically selects the correct data size (e.g., 4 or 8 bytes) and provides representation conversion in heterogeneous environments. The second method gives the user complete control over communication by exposing machine representations.

Advice to implementors. In the case where the compiler does not generate errors, nothing needs to be done to the existing interface. In the case where the compiler may generate errors, a set of overloaded functions may be used. See the paper of M. Hennecke [26]. Even if the compiler does not generate errors, explicit interfaces for all routines would be useful for detecting errors in the argument list. Also, explicit interfaces which give INTENT information can reduce the amount of copying for BUF(*) arguments. (End of advice to implementors.)
Parameterized Datatypes with Specified Precision and Exponent Range

MPI provides named datatypes corresponding to standard Fortran 77 numeric types — MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides representation conversion in heterogeneous environments. The mechanism described in this section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables are declared (perhaps indirectly) using selected_real_kind(p, r) to determine the KIND parameter, where p is decimal digits of precision and r is an exponent range. Implicitly MPI maintains a two-dimensional array of predefined MPI datatypes D(p, r). D(p, r) is defined for each value of (p, r) supported by the compiler, including pairs for which one value is unspecified. Attempting to access an element of the array with an index (p, r) not supported by the compiler is erroneous. MPI implicitly maintains a similar array of COMPLEX datatypes. For integers, there is a similar implicit array related to selected_int_kind and indexed by the requested number of digits r. Note that the predefined datatypes contained in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but a new set.

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.)

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)
IN p precision, in decimal digits (integer)
IN r decimal exponent range (integer)
OUT newtype the requested MPI datatype (handle)

int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
INTEGER P, R, NEWTYPE, IERROR
static MPI::Datatype MPI::::Create_f90_real(int p, int r)

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected_real_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r).

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.)

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)
IN p precision, in decimal digits (integer)
IN r decimal exponent range (integer)
OUT newtype the requested MPI datatype (handle)

int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
INTEGER P, R, NEWTYPE, IERROR
static MPI::Datatype MPI::::Create_f90_real(int p, int r)

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected_real_kind(p, r). In the model described above it returns a handle for the element D(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 \texttt{MPI\_UNDEFINED}. In communication, an MPI datatype \( A \) returned by \texttt{MPI\_TYPE\_CREATE\_F90\_REAL} matches a datatype \( B \) if and only if \( B \) was returned by \texttt{MPI\_TYPE\_CREATE\_F90\_REAL} called with the same values for \( p \) and \( r \) or \( B \) is a duplicate of such a datatype. Restrictions on using the returned datatype with the “external32” data representation are given on page 474.

It is erroneous to supply values for \( p \) and \( r \) not supported by the compiler.

\[ \begin{align*}
\text{MPI\_TYPE\_CREATE\_F90\_COMPLEX}(p, r, \text{newtype}) \\
\text{OUT} & \text{ newtype} & \text{the requested MPI datatype (handle)} \\
\end{align*} \]

\begin{verbatim}
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
INTEGER P, R, NEWTYPE, IERROR
static MPI\::Datatype MPI\::Datatype::Create_f90_complex(int p, int r)
static MPI\::Datatype MPI\::Datatype::Create_f90_complex(int p, int r)
\end{verbatim}

This function returns a predefined MPI datatype that matches a COMPLEX variable of \texttt{KIND selected\_real\_kind}(\( p \), \( r \)). Either \( p \) or \( r \) may be omitted from calls to \texttt{selected\_real\_kind}(\( p \), \( r \)) (but not both). Analogously, either \( p \) or \( r \) may be set to \texttt{MPI\_UNDEFINED}. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by \texttt{MPI\_TYPE\_CREATE\_F90\_REAL}. Restrictions on using the returned datatype with the “external32” data representation are given on page 474.

It is erroneous to supply values for \( p \) and \( r \) not supported by the compiler.

\[ \begin{align*}
\text{MPI\_TYPE\_CREATE\_F90\_INTEGER}(r, \text{newtype}) \\
\text{IN} & \text{ r} & \text{decimal exponent range, i.e., number of decimal digits (integer)} \\
\text{OUT} & \text{ newtype} & \text{the requested MPI datatype (handle)} \\
\end{align*} \]

\begin{verbatim}
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
INTEGER R, NEWTYPE, IERROR
static MPI\::Datatype MPI\::Datatype::Create_f90_integer(int r)
\end{verbatim}

This function returns a predefined MPI datatype that matches a INTEGER variable of \texttt{KIND selected\_int\_kind}(\( r \)). Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by \texttt{MPI\_TYPE\_CREATE\_F90\_REAL}. Restrictions on using the returned datatype with the “external32” data representation are given on page 474.

It is erroneous to supply a value for \( r \) that is not supported by the compiler.

Example:

\[ \text{MPI\_TYPE\_CREATE\_F90\_COMPLEX}(p, r, \text{newtype}) \]

\begin{verbatim}
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
INTEGER P, R, NEWTYPE, IERROR
static MPI::Datatype MPI::Datatype::Create_f90_complex(int p, int r)
\end{verbatim}

This function returns a predefined MPI datatype that matches a COMPLEX variable of \texttt{KIND selected\_real\_kind}(\( p \), \( r \)). Either \( p \) or \( r \) may be omitted from calls to \texttt{selected\_real\_kind}(\( p \), \( r \)) (but not both). Analogously, either \( p \) or \( r \) may be set to \texttt{MPI\_UNDEFINED}. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by \texttt{MPI\_TYPE\_CREATE\_F90\_REAL}. Restrictions on using the returned datatype with the “external32” data representation are given on page 474.

It is erroneous to supply values for \( p \) and \( r \) not supported by the compiler.

Example:
16.2. FORTRAN SUPPORT

integer longtype, quadtype
integer, parameter :: long = selected_int_kind(15)
integer(long) ii(10)
real(selected_real_kind(30)) x(10)
call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierr)  
call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierr)
...
call MPI_SEND(ii, 10, longtype, ...)
call MPI_SEND(x, 10, quadtype, ...)

Advice to users. The datatypes returned by the above functions are predefined datatypes. They cannot be freed; they do not need to be committed; they can be used with predefined reduction 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 routines.

If a variable was declared specifying a non-default 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_... with the same combination of (...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_... and using a hash-table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (...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 13.5.2 on page 414) or user-defined (Section 13.5.3 on page 415) 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 13.5.2 on page 414.

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:

\[
\begin{align*}
&\text{if (p > 33) or (r > 4931) then external32 representation is undefined} \\
&\text{else if (p > 15) or (r > 307) then external32_size = 16} \\
&\text{else if (p > 6) or (r > 37) then external32_size = 8} \\
&\text{else external32_size = 4}
\end{align*}
\]

For MPI_TYPE_CREATE_F90_COMPLEX; twice the size as for MPI_TYPE_CREATE_F90_REAL.
For MPI_TYPE_CREATE_F90_INTEGER:

\[
\begin{align*}
&\text{if (r > 38) then external32 representation is undefined} \\
&\text{else if (r > 18) then external32_size = 16} \\
&\text{else if (r > 4) then external32_size = 8} \\
&\text{else if (r > 2) then external32_size = 2} \\
&\text{else external32_size = 1}
\end{align*}
\]

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL and many MPI_FILE functions, when the “external32” data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each typeclass (integer, real, complex) and each word size there is a unique machine representation. For every pair (typeclass, n) supported by a compiler, MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI::<TYPE>::<n> in C and Fortran and of the form MPI::<TYPE>::<n> in C++ where <TYPE> is one of REAL, INTEGER and COMPLEX, and n is the length in bytes of the machine representation. This datatype locally matches all variables of type (typeclass, n). The list of names for such types includes:

- MPI_REAL4
- MPI_INTEGER8
- MPI_COMPLEX16

We now specify how the datatypes described in this section behave when used with the “external32” external data representation described in Section 13.5.2 on page 414.

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:

\[
\begin{align*}
&\text{if (p > 33) or (r > 4931) then external32 representation is undefined} \\
&\text{else if (p > 15) or (r > 307) then external32_size = 16} \\
&\text{else if (p > 6) or (r > 37) then external32_size = 8} \\
&\text{else external32_size = 4}
\end{align*}
\]

For MPI_TYPE_CREATE_F90_COMPLEX; twice the size as for MPI_TYPE_CREATE_F90_REAL.
For MPI_TYPE_CREATE_F90_INTEGER:

\[
\begin{align*}
&\text{if (r > 38) then external32 representation is undefined} \\
&\text{else if (r > 18) then external32_size = 16} \\
&\text{else if (r > 4) then external32_size = 8} \\
&\text{else if (r > 2) then external32_size = 2} \\
&\text{else external32_size = 1}
\end{align*}
\]

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL and many MPI_FILE functions, when the “external32” data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each typeclass (integer, real, complex) and each word size there is a unique machine representation. For every pair (typeclass, n) supported by a compiler, MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI::<TYPE>::<n> in C and Fortran and of the form MPI::<TYPE>::<n> in C++ where <TYPE> is one of REAL, INTEGER and COMPLEX, and n is the length in bytes of the machine representation. This datatype locally matches all variables of type (typeclass, n). The list of names for such types includes:

- MPI_REAL4
- MPI_INTEGER8
- MPI_COMPLEX16
One datatype is required for each representation supported by the compiler. 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. All these datatypes are predefined.

The following functions allow a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

**MPI_SIZEOF(x, size)**

IN   x       a Fortran variable of numeric intrinsic type (choice)
OUT  size    size of machine representation of that type (integer)

MPI_SIZEOF(X, SIZE, IERROR)

[type] XINTEGER SIZE, IERROR

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

Advice to users. This function is similar to the C and C++ sizeof operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (End of advice to users.)

Rationale. This function is not available in other languages because it would not be useful. (End of rationale.)

**MPI_TYPE_MATCH_SIZE(typeclass, size, type)**

IN   typeclass  generic type specifier (integer)
IN   size       size, in bytes, of representation (integer)
OUT  type      datatype with correct type, size (handle)

int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)

MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, TYPE, IERROR)

INTEGER TYPECLASS, SIZE, TYPE, IERROR

One datatype is required for each representation supported by the compiler. 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. All these datatypes are predefined.

The following functions allow a user to obtain a size-specific MPI datatype for any intrinsic Fortran type.

**MPI_SIZEOF(x, size)**

IN   x       a Fortran variable of numeric intrinsic type (choice)
OUT  size    size of machine representation of that type (integer)

MPI_SIZEOF(X, SIZE, IERROR)

[type] XINTEGER SIZE, IERROR

This function returns the size in bytes of the machine representation of the given variable. It is a generic Fortran routine and has a Fortran binding only.

Advice to users. This function is similar to the C and C++ sizeof operator but behaves slightly differently. If given an array argument, it returns the size of the base element, not the size of the whole array. (End of advice to users.)

Rationale. This function is not available in other languages because it would not be useful. (End of rationale.)

**MPI_TYPE_MATCH_SIZE(typeclass, size, type)**

IN   typeclass  generic type specifier (integer)
IN   size       size, in bytes, of representation (integer)
OUT  type      datatype with correct type, size (handle)

int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)

MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, TYPE, IERROR)

INTEGER TYPECLASS, SIZE, TYPE, IERROR
static MPI::Datatype::Match_size(int typeclass, int size)

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 MPI_SIZEOF
in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find a
suitable datatype. In C and C++, one can use the C function sizeof(), instead of
MPI_SIZEOF. 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) {
        case MPI_TYPECLASS_REAL: switch(size) {
            case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
            case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
            default: error(...);
        }
        case MPI_TYPECLASS_INTEGER: switch(size) {
            case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
            case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
            default: error(...);
        }
        ... etc ...
    }
}

(End of advice to implementors.)

Communication With Size-specific Types

The usual type matching rules apply to size-specific datatypes: a value sent with datatype
MPI_<TYPE>_n can be received with this same datatype on another process. Most modern
computers use 2’s complement for integers and IEEE format for floating point. Thus, commu-
nication using these size-specific datatypes will not entail loss of precision or truncation
errors.

Advice to users. Care is required when communicating in a heterogeneous environ-
ment. Consider the following code:

real(selected_real_kind(5)) x(100)

static MPI::Datatype::Match_size(int typeclass, int size)

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 MPI_SIZEOF
in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find a
suitable datatype. In C and C++, one can use the C function sizeof(), instead of
MPI_SIZEOF. 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) {
        case MPI_TYPECLASS_REAL: switch(size) {
            case 4: *rtype = MPI_REAL4; return MPI_SUCCESS;
            case 8: *rtype = MPI_REAL8; return MPI_SUCCESS;
            default: error(...);
        }
        case MPI_TYPECLASS_INTEGER: switch(size) {
            case 4: *rtype = MPI_INTEGER4; return MPI_SUCCESS;
            case 8: *rtype = MPI_INTEGER8; return MPI_SUCCESS;
            default: error(...);
        }
        ... etc ...
    }
}

(End of advice to implementors.)

Communication With Size-specific Types

The usual type matching rules apply to size-specific datatypes: a value sent with datatype
MPI_<TYPE>_n can be received with this same datatype on another process. Most modern
computers use 2’s complement for integers and IEEE format for floating point. Thus, commu-
nication using these size-specific datatypes will not entail loss of precision or truncation
errors.

Advice to users. Care is required when communicating in a heterogeneous environ-
ment. Consider the following code:

real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
if (myrank .eq. 0) then
  ... initialize x ...
call MPI_SEND(x, xtype, 100, 1, ...)
extif if (myrank .eq. 1) then
call MPI_RECV(x, xtype, 100, 0, ...)
endif

This may not work in a heterogeneous environment if the value of size is not the
same on process 1 and process 0. There should be no problem in a homogenous
environment. To communicate in a heterogeneous environment, there are at least four
options. The first is to declare variables of default type and use the MPI datatypes
for these types, e.g., declare a variable of type REAL and use MPI_REAL. The second
is to use selected_real_kind or selected_int_kind and with the functions of the
previous section. The third is to declare a variable that is known to be the same
size on all architectures (e.g., selected_real_kind(12) on almost all compilers will
result in an 8-byte representation). The fourth is to carefully check representation
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 at-
tention to the representation sizes. Consider the following code:

real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
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)
call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32', &
    MPI_INFO_NULL, ierror)
call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
call MPI_FILE_CLOSE(fh, ierror)
endif
call MPI_BARRIER(MPI_COMM_WORLD, ierror)
if (myrank .eq. 1) then
  call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo', &
    MPI_MODE_RDONLY, &
    MPI_INFO_NULL, fh, ierror)
call MPI_FILE_SET_VIEW(fh, 0, xtype, xtype, 'external32', &
    MPI_INFO_NULL, ierror)
call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
call MPI_FILE_CLOSE(fh, ierror)
endif
CHAPTER 16. LANGUAGE BINDINGS

16.3 Language Interoperability

16.3.1 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, 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.

There are several issues that need to be addressed in order to achieve interoperability.

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 extendable to new languages, should MPI bindings be defined for such languages.

16.3.2 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 data types 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/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 or C++ program. We also assume that Fortran, C, and C++ have address-sized 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.

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.)
16.3.3 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) \texttt{argc, argv} arguments of the C/C++ version of MPI_INIT in order to propagate values for \texttt{argc} and \texttt{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_FINALIZED 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 need perform initialization for a language only if that language library is loaded. (End of advice to implementors.)

16.3.4 Transfer of Handles
Handles are passed between Fortran and C or C++ by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C or C++ handles in Fortran. Handles are passed between C and C++ using overloaded C++ operators called from C++ code. There is no direct access to C++ objects from C.
The type definition \texttt{MPI_Fint} is provided in C/C++ for an integer of the size that matches a Fortran \texttt{INTEGER}; usually, \texttt{MPI_Fint} will be equivalent to \texttt{int}.
The following functions are provided in C to convert from a Fortran communicator handle (which is an integer) to a C communicator handle, and vice versa. See also Section 2.6.5 on page 21.

\begin{verbatim}
MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
\end{verbatim}

If \texttt{comm} is a valid Fortran handle to a communicator, then \texttt{MPI_Comm_f2c} returns a valid C handle to that same communicator; if \texttt{comm = MPI_COMM_NULL} (Fortran value), then \texttt{MPI_Comm_f2c} returns a null C handle; if \texttt{comm} is an invalid Fortran handle, then \texttt{MPI_Comm_f2c} returns an invalid C handle.

\begin{verbatim}
MPI_Fint MPI_Comm_c2f(MPI_Comm comm)
\end{verbatim}
The function \texttt{MPI_Comm_c2f} translates a C communicator handle into a Fortran handle to the same communicator; it maps a null handle into a null handle and an invalid handle into an invalid handle.
Similar functions are provided for the other types of opaque objects.

```
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)
MPI_Group MPI_Group_f2c(MPI_Fint group)
MPI_Fint MPI_Group_c2f(MPI_Group group)
MPI_Request MPI_Request_f2c(MPI_Fint request)
MPI_Fint MPI_Request_c2f(MPI_Request request)
MPI_File MPI_File_f2c(MPI_Fint file)
MPI_Fint MPI_File_c2f(MPI_File file)
MPI_WIN MPI_WIN_f2c(MPI_Fint win)
MPI_Fint MPI_WIN_c2f(MPI_WIN win)
MPI_Op MPI_Op_f2c(MPI_Fint op)
MPI_Fint MPI_Op_c2f(MPI_Op op)
MPI_Info MPI_Info_f2c(MPI_Fint info)
MPI_Fint MPI_Info_c2f(MPI_Info info)
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)
```

Example 16.13 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
! 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);
}
```
In order for a C program to use a C++ library, the C++ library must export a C interface that provides appropriate conversions before invoking the underlying C++ library call. This example shows a C interface function that invokes a C++ library call with a C communicator; the communicator is automatically promoted to a C++ handle when the underlying C++ function is invoked.

// C++ library function prototype
void cpp_lib_call(MPI::Comm cpp_comm);  

// Exported C function prototype
extern "C" {
  void c_interface(MPI_Comm c_comm);
}

void c_interface(MPI_Comm c_comm) {
    *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.)

C and C++ The C++ language interface provides the functions listed below for mixed-language interoperability. The token <CLASS> is used below to indicate any valid MPI opaque handle name (e.g., Group), except where noted. For the case where the C++ class corresponding to <CLASS> has derived classes, functions are also provided for converting between the derived classes and the C MPI::<CLASS>.

The following function allows assignment from a C MPI handle to a C++ MPI handle.

MPI::<CLASS>::operator=(const MPI::<CLASS>& data)

The constructor below creates a C++ MPI object from a C MPI handle. This allows the automatic promotion of a C MPI handle to a C++ MPI handle.

MPI::<CLASS>::<CLASS>(const MPI::<CLASS>& data)

Example 16.14 In order for a C program to use a C++ library, the C++ library must export a C interface that provides appropriate conversions before invoking the underlying C++ library call. This example shows a C interface function that invokes a C++ library call with a C communicator; the communicator is automatically promoted to a C++ handle when the underlying C++ function is invoked.

// C++ library function prototype
void cpp_lib_call(MPI::Comm cpp_comm);  

// Exported C function prototype
extern "C" {
  void c_interface(MPI_Comm c_comm);
}

void c_interface(MPI_Comm c_comm) {
    *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.)

C and C++ The C++ language interface provides the functions listed below for mixed-language interoperability. The token <CLASS> is used below to indicate any valid MPI opaque handle name (e.g., Group), except where noted. For the case where the C++ class corresponding to <CLASS> has derived classes, functions are also provided for converting between the derived classes and the C MPI::<CLASS>.

The following function allows assignment from a C MPI handle to a C++ MPI handle.

MPI::<CLASS>::operator=(const MPI::<CLASS>& data)

The constructor below creates a C++ MPI object from a C MPI handle. This allows the automatic promotion of a C MPI handle to a C++ MPI handle.

MPI::<CLASS>::<CLASS>(const MPI::<CLASS>& data)

Example 16.14 In order for a C program to use a C++ library, the C++ library must export a C interface that provides appropriate conversions before invoking the underlying C++ library call. This example shows a C interface function that invokes a C++ library call with a C communicator; the communicator is automatically promoted to a C++ handle when the underlying C++ function is invoked.

// C++ library function prototype
void cpp_lib_call(MPI::Comm cpp_comm);  

// Exported C function prototype
extern "C" {
  void c_interface(MPI_Comm c_comm);
}

void c_interface(MPI_Comm c_comm) {

The following function allows conversion from C++ objects to C MPI handles. In this case, the casting operator is overloaded to provide the functionality.

Example 16.15 A C library routine is called from a C++ program. The C library routine is prototyped to take an MPI_Comm as an argument.

Rationale. Providing conversion from C to C++ via constructors and from C++ to C via casting allows the compiler to make automatic conversions. Calling C from C++ becomes trivial, as does the provision of a C or Fortran interface to a C++ library. (End of rationale.)

Advice to users. Note that the casting and promotion operators return new handles by value. Using these new handles as INOUT parameters will affect the internal MPI object, but will not affect the original handle from which it was cast. (End of advice to users.)

It is important to note that all C++ objects and their corresponding C handles can be used interchangeably by an application. For example, an application can cache an attribute on MPI_COMM_WORLD and later retrieve it from MPI::COMM_WORLD.

16.3.5 Status

The following two procedures are provided in C to convert from a Fortran 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(MPI_Fint *f_status, MPI_Status *c_status)

    If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with
the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.

The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, respectively. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

To do the conversion in the other direction, we have the following:

int MPI_Status_c2f(MPI_Status* c_status, MPI_Fint* f_status)

This call converts a C status into a Fortran status, and has a behavior similar to MPI_Status_f2c. That is, the value of c_status must not be either MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE.

Advice to users. There is not a separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status. (End of advice to users.)

Rationale. The handling of MPI_STATUS_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUSIGNORE, then the type of its result would have to be MPI_Statust**, which was considered an inferior solution. (End of rationale.)

16.3.6 MPI Opaque Objects

Unless said otherwise, opaque objects are “the same” in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail, issues that arise for each type of MPI object.

Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.
The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see 16.3.9, page 488).

Example 16.16

```
! 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]);
    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, as returned by MPI_GET_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address is stored in the datatype, when datatypes with absolute addresses are constructed. When a send or receive operation is performed, then addresses stored in a datatype
are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly 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/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/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.)

Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated with communicators and files, attribute copy and delete functions are associated with attribute keys, reduce operations are associated with operation objects, etc. In a multilanguage environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPI implementations must make sure that such invocation will use the calling convention of the language the function is bound to.

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 is used to generate the right calling sequence when the callback function is invoked. (End of advice to implementors.)

Error Handlers

Advice to implementors. Error handlers, have, in C and C++, a "stderrarg" 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.)

Reduce Operations

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, C++, and Fortran datatypes. (End of advice to users.)

Addresses

Some of the datatype accessors and constructors have arguments of type MPI_Aint (in C) or MPI: Aint in C++, to hold addresses. The corresponding arguments, in Fortran, have type INTEGER. This causes Fortran and C/C++ to be incompatible, in an environment where addresses have 64 bits, but Fortran INTEGERs have 32 bits. This is a problem, irrespective of interlanguage issues. Suppose that a Fortran process has an address space of \( \geq 4 \) GB. What should be the value returned in Fortran by

are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly 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/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/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.)

Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated with communicators and files, attribute copy and delete functions are associated with attribute keys, reduce operations are associated with operation objects, etc. In a multilanguage environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPI implementations must make sure that such invocation will use the calling convention of the language the function is bound to.

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 is used to generate the right calling sequence when the callback function is invoked. (End of advice to implementors.)

Error Handlers

Advice to implementors. Error handlers, have, in C and C++, a "stderrarg" 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.)

Reduce Operations

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, C++, and Fortran datatypes. (End of advice to users.)

Addresses

Some of the datatype accessors and constructors have arguments of type MPI_Aint (in C) or MPI: Aint in C++, to hold addresses. The corresponding arguments, in Fortran, have type INTEGER. This causes Fortran and C/C++ to be incompatible, in an environment where addresses have 64 bits, but Fortran INTEGERs have 32 bits. This is a problem, irrespective of interlanguage issues. Suppose that a Fortran process has an address space of \( \geq 4 \) GB. What should be the value returned in Fortran by

are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly 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/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/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.)

Callback Functions

MPI calls may associate callback functions with MPI objects: error handlers are associated with communicators and files, attribute copy and delete functions are associated with attribute keys, reduce operations are associated with operation objects, etc. In a multilanguage environment, a function passed in an MPI call in one language may be invoked by an MPI call in another language. MPI implementations must make sure that such invocation will use the calling convention of the language the function is bound to.

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 is used to generate the right calling sequence when the callback function is invoked. (End of advice to implementors.)

Error Handlers

Advice to implementors. Error handlers, have, in C and C++, a "stderrarg" 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.)

Reduce Operations

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, C++, and Fortran datatypes. (End of advice to users.)

Addresses

Some of the datatype accessors and constructors have arguments of type MPI_Aint (in C) or MPI: Aint in C++, to hold addresses. The corresponding arguments, in Fortran, have type INTEGER. This causes Fortran and C/C++ to be incompatible, in an environment where addresses have 64 bits, but Fortran INTEGERs have 32 bits. This is a problem, irrespective of interlanguage issues. Suppose that a Fortran process has an address space of \( \geq 4 \) GB. What should be the value returned in Fortran by
MPI_ADDRESS, for a variable with an address above $2^{32}$? The design described here addresses this issue, while maintaining compatibility with current Fortran codes.

The constant MPI_ADDRESS_KIND is defined so that, in Fortran, INTEGER(KIND=MPI_ADDRESS_KIND) is an address sized integer type (typically, but not necessarily, the size of an INTEGER(KIND=MPI_ADDRESS_KIND) is 4 on 32 bit address machines and 8 on 64 bit address machines). Similarly, the constant MPI_INTEGER_KIND is defined so that INTEGER(KIND=MPI_INTEGER_KIND) is a default size INTEGER.

There are seven functions that have address arguments: MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB and MPI_TYPE_USB.

Four new functions are provided to supplement the first four functions in this list. These functions are described in Section 4.1.1 on page 79. The remaining three functions are supplemented by the new function MPI_TYPE_GET_EXTENT, described in that same section. The new functions have the same functionality as the old functions in C/C++, or on Fortran systems where default INTEGERs are address sized. In Fortran, they accept arguments of type INTEGER(KIND=MPI_ADDRESS_KIND), wherever arguments of type MPI:aint and MPI:aint are used in C and C++. On Fortran 77 systems 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 an appropriate integer type. The old functions will continue to be provided, for backward compatibility. However, users are encouraged to switch to the new functions, in Fortran, so as to avoid problems on systems with an address range $> 2^{32}$, and to provide compatibility across languages.

### 16.3.7 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, MPI_TAG, etc.)

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPL_{TYPE,COMM,WIN}::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,” “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 6.7 on page 221 define attributes to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C/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/C++ callee, or vice-versa.

MPI will store, internally, address sized attributes. If Fortran INTEGERs are smaller, then the Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the 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/C++. These functions are described in Section 6.7, page 221. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integer valued attributes. C and 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 or C++, then \texttt{MPI\_xxx\_get\_attr} will return the address of (a pointer to) the integer valued attribute. When an address valued attribute is accessed from Fortran, then \texttt{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 \texttt{MPI\_ADDRESS\_KIND} is returned. The conversion may cause truncation if deprecated attribute functions are used.

\textbf{Example 16.17} A. C to Fortran

\begin{verbatim}
C code
static int i = 5;
void *p;
p = &i;
MPI_Comm_put_attr(..., p);
....

Fortran code
INTEGER(kind = MPI_ADDRESS_KIND) val
CALL MPI_COMM_GET_ATTR(..., val,...)
IF(val.NE.address_of_i) THEN CALL ERROR

B. Fortran to C

Fortran code
INTEGER(kind=MPI_ADDRESS_KIND) val
val = 55555
CALL MPI_COMM_PUT_ATTR(..., val,ierr)

C code
int *p;
MPI_Comm_get_attr(...,&p,...);
if (*p != 55555) error();
\end{verbatim}

The predefined MPI attributes can be integer valued or address valued. Predefined integer valued attributes, such as \texttt{MPI\_TAG\_UB}, behave as if they were put by a Fortran call, i.e., in Fortran, \texttt{MPI\_COMM\_GET\_ATTR(MPI\_COMM\_WORLD, MPI\_TAG\_UB, val, 123456789 1011112131415161718192021222324252627282930313233343536373839404142434445464748}}
flag, ierr) will return in val the upper bound for tag value; in C,
MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UNDEFINED, &p, &flag) will return in p a
pointer to an int containing the upper bound for tag value.

Address valued predefined attributes, such as MPI_WIN_BASE, behave as if they were
put by a C call, i.e., in Fortran, MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, val, flag,
ierror) will return in val the base address of the window, converted to an integer. In C,
MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window
base, cast to (void *).

Rationale. The design is consistent with the behavior specified for predefined at-
tributes, and ensures that no information is lost when attributes are passed from
language to language. (End of rationale.)

Advice to implementors. Implementations should tag attributes either as address
attributes or as integer attributes, according to whether they were set in C or in
Fortran. Thus, the right choice can be made when the attribute is retrieved. (End of
decision to implementors.)

16.3.8 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.

16.3.9 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 16.3.4. Constants that
specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in
Fortran than in C/C++ since in C/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/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 must be in Fortran 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 ...). Requiring that the Fortran and C values be the same will complicate the initialization process. *(End of rationale.)*

### 16.3.10 Interlanguage Communication

The type matching rules for communications 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 16.18** In the example below, a Fortran array is sent from Fortran and received in C.

```fortran
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, IERR, MYRANK, 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 MPI_TYPE_COMMIT(TYPE, IERR)
CALL MPI_COMM_RANK( MPI_COMM_WORLD, MYRANK, IERR)
IF (MYRANK.EQ.0) THEN
  CALL MPI_SEND( MPI_BOTTOM, 1, TYPE, 1, 0, MPI_COMM_WORLD, IERR)
ELSE
  CALL C_ROUTINE(TYPE)
END IF
/
```

```c
void C_ROUTINE(MPI_Fint *fhandle)
```

---

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 must be in Fortran 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 ...). Requiring that the Fortran and C values be the same will complicate the initialization process. *(End of rationale.)*
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., 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.
**Annex A**

**Language Bindings Summary**

In this section we summarize the specific bindings for C, Fortran, and C++. 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 name is listed in the left column and the C++ name is listed in the middle or right column.

<table>
<thead>
<tr>
<th>Return Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ type: const int (or unnamed enum)</td>
</tr>
<tr>
<td>MPI_SUCCESS</td>
</tr>
<tr>
<td>MPI_ERRBUFFER</td>
</tr>
<tr>
<td>MPI_ERRCOUNT</td>
</tr>
<tr>
<td>MPI_ERRTYPE</td>
</tr>
<tr>
<td>MPI_ERR_TAG</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
</tr>
<tr>
<td>MPI_ERR_REQUEST</td>
</tr>
<tr>
<td>MPI_ERR_ROOT</td>
</tr>
<tr>
<td>MPI_ERR_GROUP</td>
</tr>
<tr>
<td>MPI_ERR_OP</td>
</tr>
<tr>
<td>MPI_ERR_TOPOLOGY</td>
</tr>
<tr>
<td>MPI_ERR_DIMS</td>
</tr>
<tr>
<td>MPI_ERR_ARG</td>
</tr>
<tr>
<td>MPI_ERR_UNKNOWN</td>
</tr>
<tr>
<td>MPI_ERR_TRUNCATE</td>
</tr>
<tr>
<td>MPI_ERR_OTHER</td>
</tr>
<tr>
<td>MPI_ERR_IN_STATUS</td>
</tr>
</tbody>
</table>

(Continued on next page)
<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERR_ACCESS</td>
<td>Access Error</td>
</tr>
<tr>
<td>MPI_ERR_AMODE</td>
<td>AMODE Error</td>
</tr>
<tr>
<td>MPI_ERR_ASSERT</td>
<td>Assert Error</td>
</tr>
<tr>
<td>MPI_ERR_BAD_FILE</td>
<td>Bad File Error</td>
</tr>
<tr>
<td>MPI_ERR_BASE</td>
<td>Base Error</td>
</tr>
<tr>
<td>MPI_ERR_CONVERSION</td>
<td>Conversion Error</td>
</tr>
<tr>
<td>MPI_ERR_DISP</td>
<td>Disp Error</td>
</tr>
<tr>
<td>MPI_ERR_dup_datarep</td>
<td>Dup Datarep Error</td>
</tr>
<tr>
<td>MPI_ERR_FILE_EXISTS</td>
<td>File Exists Error</td>
</tr>
<tr>
<td>MPI_ERR_FILE_IN_USE</td>
<td>File In Use Error</td>
</tr>
<tr>
<td>MPI_ERR_FILE</td>
<td>File Error</td>
</tr>
<tr>
<td>MPI_ERR_INFO_KEY</td>
<td>Info Key Error</td>
</tr>
<tr>
<td>MPI_ERR_INFO_NOKEY</td>
<td>Info Nokey Error</td>
</tr>
<tr>
<td>MPI_ERR_INFO_VALUE</td>
<td>Info Value Error</td>
</tr>
<tr>
<td>MPI_ERR_INFO</td>
<td>Info Error</td>
</tr>
<tr>
<td>MPI_ERR_IO</td>
<td>Io Error</td>
</tr>
<tr>
<td>MPI_ERR_KEYVAL</td>
<td>Keyval Error</td>
</tr>
<tr>
<td>MPI_ERR_LOCKTYPE</td>
<td>Locktype Error</td>
</tr>
<tr>
<td>MPI_ERR_NAME</td>
<td>Name Error</td>
</tr>
<tr>
<td>MPI_ERR_NO_MEM</td>
<td>No Mem Error</td>
</tr>
<tr>
<td>MPI_ERR_NOTSAME</td>
<td>Not Same Error</td>
</tr>
<tr>
<td>MPI_ERR_NO_SPACE</td>
<td>No Space Error</td>
</tr>
<tr>
<td>MPI_ERR_NO_SUCH_FILE</td>
<td>No Such File Error</td>
</tr>
<tr>
<td>MPI_ERR_PORT</td>
<td>Port Error</td>
</tr>
<tr>
<td>MPI_ERR_QUOTA</td>
<td>Quota Error</td>
</tr>
<tr>
<td>MPI_ERR_READ_ONLY</td>
<td>Read Only Error</td>
</tr>
<tr>
<td>MPI_ERR_RMA_CONFLICT</td>
<td>Rma Conflict Error</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SYNC</td>
<td>Rma Sync Error</td>
</tr>
<tr>
<td>MPI_ERR_SERVICE</td>
<td>Service Error</td>
</tr>
<tr>
<td>MPI_ERR_SIZE</td>
<td>Size Error</td>
</tr>
<tr>
<td>MPI_ERR_SPAWN</td>
<td>Spawn Error</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_DATAREP</td>
<td>Unsupported Datarep Error</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_OPERATION</td>
<td>Unsupported Operation Error</td>
</tr>
<tr>
<td>MPI_ERR_WIN</td>
<td>Win Error</td>
</tr>
<tr>
<td>MPI_ERR_LASTCODE</td>
<td>Lastcode Error</td>
</tr>
</tbody>
</table>
### Assorted Constants

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_BOTTOM</td>
<td>MPI::BOTTOM</td>
<td>void * const</td>
</tr>
<tr>
<td>MPI_PROC_NULL</td>
<td>MPI::PROC_NULL</td>
<td>const int</td>
</tr>
<tr>
<td>MPI_ANY_SOURCE</td>
<td>MPI::ANY_SOURCE</td>
<td>(or unnamed enum)</td>
</tr>
<tr>
<td>MPI_UNDEFINED</td>
<td>MPI::UNDEFINED</td>
<td></td>
</tr>
<tr>
<td>MPI_BSEND_OVERHEAD</td>
<td>MPI::BSEND_OVERHEAD</td>
<td></td>
</tr>
<tr>
<td>MPI_KEYVAL_INVALID</td>
<td>MPI::KEYVAL_INVALID</td>
<td></td>
</tr>
<tr>
<td>MPI_INPLACE</td>
<td>MPI::INPLACE</td>
<td></td>
</tr>
<tr>
<td>MPI_LOCK_EXCLUSIVE</td>
<td>MPI::LOCK_EXCLUSIVE</td>
<td></td>
</tr>
<tr>
<td>MPI_LOCK_SHARED</td>
<td>MPI::LOCK_SHARED</td>
<td></td>
</tr>
<tr>
<td>MPI_ROOT</td>
<td>MPI::ROOT</td>
<td></td>
</tr>
</tbody>
</table>

### Status size and reserved index values (Fortran only)

<table>
<thead>
<tr>
<th>MPI_STATUS_SIZE</th>
<th>Not defined for C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_TAG</td>
<td>Not defined for C++</td>
</tr>
<tr>
<td>MPI_ERROR</td>
<td>Not defined for C++</td>
</tr>
</tbody>
</table>

### Variable Address Size (Fortran only)

<table>
<thead>
<tr>
<th>MPI_ADDRESS_KIND</th>
<th>Not defined for C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER_KIND</td>
<td>Not defined for C++</td>
</tr>
<tr>
<td>MPI_OFFSET_KIND</td>
<td>Not defined for C++</td>
</tr>
</tbody>
</table>

### Error-handling specifiers

<table>
<thead>
<tr>
<th>C++ type: MPI::Errhandler</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERRORS_ARE_FATAL</td>
</tr>
<tr>
<td>MPI_ERRORS_RETURN</td>
</tr>
</tbody>
</table>

### Maximum Sizes for Strings

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX_PROCESSOR_NAME</td>
<td>MPI::MAX_PROCESSOR_NAME</td>
<td>const int</td>
</tr>
<tr>
<td>MPI_MAX_ERROR_STRING</td>
<td>MPI::MAX_ERROR_STRING</td>
<td>(or unnamed enum)</td>
</tr>
<tr>
<td>MPI_MAX_DAIREP_STRING</td>
<td>MPI::MAX_DAIREP_STRING</td>
<td></td>
</tr>
<tr>
<td>MPI_MAX_INFO_KEY</td>
<td>MPI::MAX_INFO_KEY</td>
<td></td>
</tr>
<tr>
<td>MPI_MAX_INFO_VAL</td>
<td>MPI::MAX_INFO_VAL</td>
<td></td>
</tr>
<tr>
<td>MPI_MAX_OBJECT_NAME</td>
<td>MPI::MAX_OBJECT_NAME</td>
<td></td>
</tr>
<tr>
<td>MPI_MAX_PORT_NAME</td>
<td>MPI::MAX_PORT_NAME</td>
<td></td>
</tr>
</tbody>
</table>
## Annexe A. Language bindings summary

### C and C++ types

<table>
<thead>
<tr>
<th>C/C++ types</th>
<th>Named Predefined Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>char</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>signed short int</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>signed int</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>signed long</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>signed long long</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>signed char</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>(treated as integral value)</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_UCHAR</td>
</tr>
<tr>
<td>(treated as printable character)</td>
<td>MPI_UCHAR</td>
</tr>
<tr>
<td>signed short int</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>signed int</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>unsigned short</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>unsigned long</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>(synonym)</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>(synonym)</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>(synonym)</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>unsigned long</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>(synonym)</td>
<td>MPI_USHORT</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>MPI_LONG</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>(defined in &lt;stddef.h&gt;)</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>(any C/C++ type)</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_SHORT</td>
</tr>
<tr>
<td>(treated as printable character)</td>
<td>MPI_SHORT</td>
</tr>
<tr>
<td>double</td>
<td>MPI_DOUBLE</td>
</tr>
<tr>
<td>long double</td>
<td>MPI_DOUBLE</td>
</tr>
<tr>
<td>wchar_t</td>
<td>MPI_WCHAR</td>
</tr>
<tr>
<td>(treated as printable character)</td>
<td>MPI_WCHAR</td>
</tr>
<tr>
<td>REAL</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>byte</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>(any C/C++ type)</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>PACKED</td>
<td>MPI_PACKED</td>
</tr>
<tr>
<td>(any C/C++ type)</td>
<td>MPI_PACKED</td>
</tr>
</tbody>
</table>

### Fortran types

<table>
<thead>
<tr>
<th>Fortran types</th>
<th>Named Predefined Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>REAL</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>CHARACTER(1)</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>BYTE</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>PACKED</td>
<td>MPI_CHAR</td>
</tr>
</tbody>
</table>

### C and C++ (no Fortran) Named Predefined Datatypes

<table>
<thead>
<tr>
<th>Named Predefined Datatypes</th>
<th>C++ type: MPI::Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>MPI_COMPLEX</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>CHARACTER(1)</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>BYTE</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>PACKED</td>
<td>MPI_PACKED</td>
</tr>
</tbody>
</table>

### Fortran types

<table>
<thead>
<tr>
<th>Fortran types</th>
<th>Named Predefined Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>REAL</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>MPI_COMPLEX</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>CHARACTER(1)</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>BYTE</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>PACKED</td>
<td>MPI_PACKED</td>
</tr>
</tbody>
</table>
A.1. DEFINED VALUES AND HANDLES

C++-Only Named Predefined Datatypes

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::BOOL</td>
<td>bool</td>
</tr>
<tr>
<td>MPI::COMPLEX</td>
<td>Complex&lt;float&gt;</td>
</tr>
<tr>
<td>MPI::DOUBLE_COMPLEX</td>
<td>Complex&lt;double&gt;</td>
</tr>
<tr>
<td>MPI::LONG_DOUBLE_COMPLEX</td>
<td>Complex&lt;long double&gt;</td>
</tr>
</tbody>
</table>

Optional datatypes (Fortran)

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>Fortran types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::DOUBLE</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td>MPI::INTEGER1</td>
<td>INTEGER*1</td>
</tr>
<tr>
<td>MPI::INTEGER2</td>
<td>INTEGER*2</td>
</tr>
<tr>
<td>MPI::INTEGER4</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>MPI::INTEGER8</td>
<td>INTEGER*8</td>
</tr>
<tr>
<td>MPI::REAL2</td>
<td>REAL*2</td>
</tr>
<tr>
<td>MPI::REAL4</td>
<td>REAL*4</td>
</tr>
<tr>
<td>MPI::REAL8</td>
<td>REAL*8</td>
</tr>
</tbody>
</table>

Datatypes for reduction functions (C and C++)

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_FLOAT_INT</td>
<td>MPI_FLOAT_INT</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>MPI_DOUBLE_INT</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>MPI_LONG_INT</td>
</tr>
<tr>
<td>MPI_2INT</td>
<td>MPI_2INT</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>MPI_SHORT_INT</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>MPI_LONG_DOUBLE_INT</td>
</tr>
</tbody>
</table>

Datatypes for reduction functions (Fortran)

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>Fortran types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_FLOAT_INT</td>
<td>MPI_FLOAT_INT</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>MPI_DOUBLE_INT</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>MPI_LONG_INT</td>
</tr>
<tr>
<td>MPI_2INT</td>
<td>MPI_2INT</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>MPI_SHORT_INT</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>MPI_LONG_DOUBLE_INT</td>
</tr>
</tbody>
</table>

Special datatypes for constructing derived datatypes

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::UB</td>
<td>MPI::UB</td>
</tr>
<tr>
<td>MPI::LB</td>
<td>MPI::LB</td>
</tr>
</tbody>
</table>

Reserved communicators

<table>
<thead>
<tr>
<th>C++ type: MPI::Intracomm</th>
<th>C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::COMM_WORLD</td>
<td>MPI::COMM_WORLD</td>
</tr>
<tr>
<td>MPI::COMM_WORLD</td>
<td>MPI::COMM_WORLD</td>
</tr>
<tr>
<td>MPI::COMM_SELF</td>
<td>MPI::COMM_SELF</td>
</tr>
</tbody>
</table>
### ANNEX A. LANGUAGE BINDINGS SUMMARY

#### Results of communicator and group comparisons

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_GROUP_NULL</td>
<td>MPI::GROUP_NULL</td>
<td>const MPI::Group</td>
</tr>
<tr>
<td>MPI_COMM_NULL</td>
<td>MPI::COMM_NULL</td>
<td>const MPI::Group</td>
</tr>
<tr>
<td>MPI_DATATYPE_NULL</td>
<td>MPI::DATATYPE_NULL</td>
<td>const MPI::Datatype</td>
</tr>
<tr>
<td>MPI_REQUEST_NULL</td>
<td>MPI::REQUEST_NULL</td>
<td>const MPI::Request</td>
</tr>
<tr>
<td>MPI_OP_NULL</td>
<td>MPI::OP_NULL</td>
<td>const MPI::Op</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_NULL</td>
<td>MPI::ERRHANDLER_NULL</td>
<td>const MPI::Errhandler</td>
</tr>
<tr>
<td>MPI_FILE_NULL</td>
<td>MPI::FILE_NULL</td>
<td></td>
</tr>
<tr>
<td>MPI_INFO_NULL</td>
<td>MPI::INFO_NULL</td>
<td></td>
</tr>
<tr>
<td>MPI_WIN_NULL</td>
<td>MPI::WIN_NULL</td>
<td></td>
</tr>
</tbody>
</table>

#### Environmental inquiry keys

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_TAG_UB</td>
<td>MPI::TAG_UB</td>
<td>const int</td>
</tr>
<tr>
<td>MPI_IO</td>
<td>MPI::IO</td>
<td></td>
</tr>
<tr>
<td>MPI_HOST</td>
<td>MPI::HOST</td>
<td></td>
</tr>
<tr>
<td>MPI_WTIME_IS_GLOBAL</td>
<td>MPI::WTIME_IS_GLOBAL</td>
<td></td>
</tr>
</tbody>
</table>

#### Collective Operations

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>MPI::MAX</td>
<td>const MPI::Op</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>MPI::MIN</td>
<td></td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>MPI::SUM</td>
<td></td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>MPI::PROD</td>
<td></td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>MPI::MAXLOC</td>
<td></td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>MPI::MINLOC</td>
<td></td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>MPI::BAND</td>
<td></td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>MPI::BOR</td>
<td></td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>MPI::BXOR</td>
<td></td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>MPI::LAND</td>
<td></td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>MPI::LOR</td>
<td></td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>MPI::LXOR</td>
<td></td>
</tr>
<tr>
<td>MPI_REPLACE</td>
<td>MPI::REPLACE</td>
<td></td>
</tr>
</tbody>
</table>

#### Null Handles

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_GROUP_NULL</td>
<td>MPI::GROUP_NULL</td>
<td>const MPI::Group</td>
</tr>
<tr>
<td>MPI_COMM_NULL</td>
<td>MPI::COMM_NULL</td>
<td>const MPI::Group</td>
</tr>
<tr>
<td>MPI_DATATYPE_NULL</td>
<td>MPI::DATATYPE_NULL</td>
<td>const MPI::Datatype</td>
</tr>
<tr>
<td>MPI_REQUEST_NULL</td>
<td>MPI::REQUEST_NULL</td>
<td>const MPI::Request</td>
</tr>
<tr>
<td>MPI_OP_NULL</td>
<td>MPI::OP_NULL</td>
<td>const MPI::Op</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_NULL</td>
<td>MPI::ERRHANDLER_NULL</td>
<td>const MPI::Errhandler</td>
</tr>
<tr>
<td>MPI_FILE_NULL</td>
<td>MPI::FILE_NULL</td>
<td></td>
</tr>
<tr>
<td>MPI_INFO_NULL</td>
<td>MPI::INFO_NULL</td>
<td></td>
</tr>
<tr>
<td>MPI_WIN_NULL</td>
<td>MPI::WIN_NULL</td>
<td></td>
</tr>
</tbody>
</table>

1) C++ type: See Section 16.1.7 on page 455 regarding class hierarchy and the specific type of MPI::COMM_NULL
### Empty group

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_GROUP_EMPTY</td>
<td>MPI::GROUP_EMPTY</td>
<td></td>
</tr>
</tbody>
</table>

### Topologies

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_GRAPH</td>
<td>MPI::GRAPH</td>
<td></td>
</tr>
<tr>
<td>MPI_CART</td>
<td>MPI::CART</td>
<td></td>
</tr>
</tbody>
</table>

### Predefined functions

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_NULL_COPY_FN</td>
<td>MPI::NULL_COPY_FN</td>
<td>MPI::Copy_function</td>
</tr>
<tr>
<td>MPI_DUP_FN</td>
<td>MPI::DUP_FN</td>
<td>MPI::Copy_function</td>
</tr>
<tr>
<td>MPI_NULL_DELETE_FN</td>
<td>MPI::NULL_DELETE_FN</td>
<td>MPI::Delete_function</td>
</tr>
</tbody>
</table>

### Predefined Attribute Keys

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_APPNUM</td>
<td>MPI::APPNUM</td>
<td></td>
</tr>
<tr>
<td>MPI_LASTUSEDCODE</td>
<td>MPI::LASTUSEDCODE</td>
<td></td>
</tr>
<tr>
<td>MPI_UNIVERSE_SIZE</td>
<td>MPI::UNIVERSE_SIZE</td>
<td></td>
</tr>
<tr>
<td>MPI_WIN_BASE</td>
<td>MPI::WIN_BASE</td>
<td></td>
</tr>
<tr>
<td>MPI_WIN_DISP_UNIT</td>
<td>MP</td>
<td>MPI::WIN_DISP_UNIT</td>
</tr>
<tr>
<td>MPI_WIN_SIZE</td>
<td>MPI::WIN_SIZE</td>
<td></td>
</tr>
</tbody>
</table>

### Mode Constants

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MODE_APPEND</td>
<td>MPI::MODE_APPEND</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_CREATE</td>
<td>MPI::MODE_CREATE</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_DELETE_ON_CLOSE</td>
<td>MPI::MODE_DELETE_ON_CLOSE</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_EXCL</td>
<td>MPI::MODE_EXCL</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_NOCHECK</td>
<td>MPI::MODE_NOCHECK</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_NOPRECEDE</td>
<td>MPI::MODE_NOPRECEDE</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_NOPUT</td>
<td>MPI::MODE_NOPUT</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_NOSTORE</td>
<td>MPI::MODE_NOSTORE</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_NOSUCCEED</td>
<td>MPI::MODE_NOSUCCEED</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_RDONLY</td>
<td>MPI::MODE_RDONLY</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_RDWR</td>
<td>MPI::MODE_RDWR</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_SEQUENTIAL</td>
<td>MPI::MODE_SEQUENTIAL</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_UNIQUE_OPEN</td>
<td>MPI::MODE_UNIQUE_OPEN</td>
<td></td>
</tr>
<tr>
<td>MPI_MODE_WONLY</td>
<td>MPI::MODE_WONLY</td>
<td></td>
</tr>
</tbody>
</table>
### Datatype Decoding Constants

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>COMBINER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTIGUOUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DARRAY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F90_INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F90_REAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HINDEXED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HINDEXED_BLOCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAMED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESIZED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRUCT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBARRAY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VECTOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Threads Constants

<table>
<thead>
<tr>
<th>Threads</th>
<th>MPITHREAD_FUNNELED</th>
<th>MPI::THREAD_FUNNELED</th>
<th>MPITHREAD_MULTIPLE</th>
<th>MPI::THREAD_MULTIPLE</th>
<th>MPITHREAD_SERIALIZED</th>
<th>MPI::THREAD_SERIALIZED</th>
<th>MPITHREAD_SINGLE</th>
<th>MPI::THREAD_SINGLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNNELED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULTIPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### File Operation Constants

<table>
<thead>
<tr>
<th>File Operation</th>
<th>MPI:DISPLACEMENT_CURRENT</th>
<th>MPI::DISPLACEMENT_CURRENT</th>
<th>MPI:DISTRIBUTE_BLOCK</th>
<th>MPI::DISTRIBUTE_BLOCK</th>
<th>MPI:DISTRIBUTE_CYCLIC</th>
<th>MPI::DISTRIBUTE_CYCLIC</th>
<th>MPI:DISTRIBUTE_DFLT_DARG</th>
<th>MPI::DISTRIBUTE_DFLT_DARG</th>
<th>MPI:ORDER_C</th>
<th>MPI::ORDER_C</th>
<th>MPI:ORDER_FORTRAN</th>
<th>MPI::ORDER_FORTRAN</th>
<th>MPI:SEEK_END</th>
<th>MPI::SEEK_END</th>
<th>MPI:SEEK_SET</th>
<th>MPI::SEEK_SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFLT_DARG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDER_C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDER_FORTRAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEEK_END</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEEK_SET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### F90 Datatype Matching Constants

<table>
<thead>
<tr>
<th>F90 Datatype</th>
<th>MPI:TYPECLASS_COMPLEX</th>
<th>MPI::TYPECLASS_COMPLEX</th>
<th>MPI:TYPECLASS_INTEGER</th>
<th>MPI::TYPECLASS_INTEGER</th>
<th>MPI:TYPECLASS_REAL</th>
<th>MPI::TYPECLASS_REAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.1. DEFINED VALUES AND HANDLES

Handles to Assorted Structures in C and C++ (no Fortran)

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_File</td>
<td>MPI::File</td>
</tr>
<tr>
<td>MPI_Info</td>
<td>MPI::Info</td>
</tr>
<tr>
<td>MPI_Win</td>
<td>MPI::Win</td>
</tr>
</tbody>
</table>

Constants Specifying Empty or Ignored Input

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ARGVS_NULL</td>
<td>MPI::ARGVS_NULL</td>
</tr>
<tr>
<td>MPI_ARGV_NULL</td>
<td>MPI::ARGV_NULL</td>
</tr>
<tr>
<td>MPI_ERRCODES_IGNORE</td>
<td>Not defined for C++</td>
</tr>
<tr>
<td>MPI_STATUSES_IGNORE</td>
<td>Not defined for C++</td>
</tr>
</tbody>
</table>

C Constants Specifying Ignored Input (no C++ or Fortran)

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_F_STATUSES_IGNORE</td>
<td>Not defined for C++</td>
</tr>
<tr>
<td>MPI_F_STATUS_IGNORE</td>
<td>Not defined for C++</td>
</tr>
</tbody>
</table>

C and C++ preprocessor Constants and Fortran Parameters

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUBVERSION</td>
<td>MPI::SUBVERSION</td>
</tr>
<tr>
<td>MPI_VERSION</td>
<td>MPI::VERSION</td>
</tr>
</tbody>
</table>

A.1.2 Types

The following are defined C type definitions, included in the file mpi.h.

```c
/* C opaque types */
MPI_Aint
MPI_Fint
MPI_Offset
MPI_Status

/* C handles to assorted structures */
MPI_Comm
MPI_Datatype
MPI_Errhandler
MPI_File
MPI_Group
MPI_Info
MPI_Offset
MPI_Request
MPI_Win

// C++ opaque types (all within the MPI namespace)
MPI::Aint
MPI::Offset
MPI::Status
```
A.1.3 Prototype definitions

The following are defined C typedefs for user-defined functions, also included in the file mpi.h.

/* prototypes for user-defined functions */
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
                             MPI_Datatype *datatype);
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm,
                                        int comm_keyval, void *extra_state, void *attribute_val_in,
                                        void *attribute_val_out, int *flag);
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm,
                                         int comm_keyval, void *attribute_val, void *extra_state);
typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
                                         void *extra_state, void *attribute_val_in,
                                         void *attribute_val_out, int *flag);
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
                                         void *attribute_val, void *extra_state);
typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
                                          int type_keyval, void *extra_state,
                                          void *attribute_val_in, void *attribute_val_out, int *flag);
typedef int MPI_Type_delete_attr_function(MPI_Datatype type,
                                          int type_keyval, void *attribute_val, void *extra_state);
typedef void MPI_Comm_errhandler_fn(MPI_Comm *, int *, ...);}
typedef void MPI_Win_errhandler_fn(MPI_Win *, int *, ...);
typedef void MPI_File_errhandler_fn(MPI_File *, int *, ...);
typedef int MPI_Grequest_query_function(void *extra_state,
                                       MPI_Status *status);
typedef int MPI_Grequest_free_function(void *extra_state);
typedef int MPI_Grequest_cancel_function(void *extra_state, int complete);
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
                                         MPI_Aint *file_extent, void *extra_state);
typedef int MPI_Datarep_conversion_function(void *userbuf,
                                           MPI_Datatype datatype, int count, void *filebuf,
                                           MPI_Offset position, void *extra_state);

For Fortran, here are examples of how each of the user-defined subroutines should be declared.

The user-function argument to MPI_OP_CREATE should be declared like this:

```fortran
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
  <type> INVEC(LEN), INOUTVEC(LEN)
  INTEGER LEN, TYPE
END SUBROUTINE USER_FUNCTION
```

The copy and delete function arguments to MPI_COMM_KEYVAL_CREATE should be declared like these:

```fortran
SUBROUTINE COMM_COPY_ATTR_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE,
                               ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
  INTEGER OLDCOMM, COMM_KEYVAL, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                              ATTRIBUTE_VAL_OUT
  LOGICAL FLAG
END SUBROUTINE COMM_COPY_ATTR_FN

SUBROUTINE COMM_DELETE_ATTR_FN(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
                                EXTRA_STATE, IERROR)
  INTEGER COMM, COMM_KEYVAL, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
END SUBROUTINE COMM_DELETE_ATTR_FN
```

The copy and delete function arguments to MPI_WIN_KEYVAL_CREATE should be declared like these:

```fortran
SUBROUTINE WIN_COPY_ATTR_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE,
                             ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
  INTEGER OLDWIN, WIN_KEYVAL, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
                              ATTRIBUTE_VAL_OUT
  LOGICAL FLAG
END SUBROUTINE WIN_COPY_ATTR_FN

SUBROUTINE WIN_DELETE_ATTR_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
                               EXTRA_STATE, IERROR)
  INTEGER WIN, WIN_KEYVAL, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
END SUBROUTINE WIN_DELETE_ATTR_FN
```
The copy and delete function arguments to MPI\_TYPE\_KEYVAL\_CREATE should be declared like these:

```fortran
SUBROUTINE TYPE_COPY_ATTR_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE,
    ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
  INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE,
    ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
  LOGICAL FLAG
END SUBROUTINE TYPE_COPY_ATTR_FN

SUBROUTINE TYPE_DELETE_ATTR_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
    EXTRA_STATE, IERROR)
  INTEGER TYPE, TYPE_KEYVAL, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
END SUBROUTINE TYPE_DELETE_ATTR_FN
```

The handler-function argument to MPI\_COMM\_CREATE\_ERRHANDLER should be declared like this:

```fortran
SUBROUTINE COMM_ERRHANDLER_FN(COMM, ERROR_CODE, ...)
  INTEGER COMM, ERROR_CODE
END SUBROUTINE COMM_ERRHANDLER_FN
```

The handler-function argument to MPI\_WIN\_CREATE\_ERRHANDLER should be declared like this:

```fortran
SUBROUTINE WIN_ERRHANDLER_FN(WIN, ERROR_CODE, ...)
  INTEGER WIN, ERROR_CODE
END SUBROUTINE WIN_ERRHANDLER_FN
```

The handler-function argument to MPI\_FILE\_CREATE\_ERRHANDLER should be declared like this:

```fortran
SUBROUTINE FILE_ERRHANDLER_FN(FILE, ERROR_CODE, ...)
  INTEGER FILE, ERROR_CODE
END SUBROUTINE FILE_ERRHANDLER_FN
```

The query, free, and cancel function arguments to MPI\_GREQUEST\_START should be declared like these:

```fortran
SUBROUTINE GREQUEST_QUERY_FUNCTION(EXTRA_STATE, STATUS, IERROR)
  INTEGER STATUS(MPI_STATUS_SIZE), IERROR
END SUBROUTINE GREQUEST_QUERY_FUNCTION

SUBROUTINE GREQUEST_FREE_FUNCTION(EXTRA_STATE, IERROR)
  INTEGER IERROR
END SUBROUTINE GREQUEST_FREE_FUNCTION

SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
  INTEGER IERROR
  LOGICAL COMPLETE
END SUBROUTINE GREQUEST_CANCEL_FUNCTION
```
A.1. DEFINED VALUES AND HANDLES

The extend and conversion function arguments to MPI\_REGISTER\_DATAREP should be declared like these:

```fortran
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
SUBROUTINE DATAREP_CONVERSION_FUNCTION(USERBUF, DATATYPE, COUNT, FILEBUF,
POSITION, EXTRA_STATE, IERROR)
<TYPE> USERBUF(*), FILEBUF(*)
INTEGER COUNT, DATATYPE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) POSITION
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

The following are defined C++ typedefs, also included in the file \texttt{mpi.h}.

```cpp
namespace MPI {
    typedef void User_function(const void* invec, void *inoutvec,
    int len, const Datatype& datatype);
    typedef int Comm::Copy_attr_function(const Comm& oldcomm,
    int comm_keyval, void* extra_state, void* attribute_val_in,
    void* attribute_val_out, bool& flag);
    typedef int Comm::Delete_attr_function(Comm& comm, int
    comm_keyval, void* attribute_val, void* extra_state);
    typedef int Win::Copy_attr_function(const Win& oldwin,
    int win_keyval, void* extra_state, void* attribute_val_in,
    void* attribute_val_out, bool& flag);
    typedef int Win::Delete_attr_function(Win& win, int
    win_keyval, void* attribute_val, void* extra_state);
    typedef int Datatype::Copy_attr_function(const Datatype& oldtype,
    int type_keyval, void* extra_state, const void* attribute_val_in,
    void* attribute_val_out, bool& flag);
    typedef int Datatype::Delete_attr_function(Datatype& type,
    int type_keyval, void* attribute_val, void* extra_state);
    typedef void Comm::Errhandler_fn(Comm &, int *, ...);
    typedef void Win::Errhandler_fn(Win &, int *, ...);
    typedef void File::Errhandler_fn(File &, int *, ...);
    typedef int Grequest::Query_function(void* extra_state, Status& status);
    typedef int Grequest::Free_function(void* extra_state);
    typedef int Grequest::Cancel_function(void* extra_state, bool complete);
    typedef void Datarep_extent_function(const Datatype& datatype,
    Aint& file_extent, void* extra_state);
}
```
typedef void Datarep_conversion_function(void* userbuf, Datatype& datatype, int count, void* filebuf, Offset position, void* extra_state);

A.1.4 Deprecated prototype definitions

The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.

/* prototypes for user-defined functions */
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag);
typedef int MPI_Delete_function(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state);
typedef void MPI_Handler_function(MPI_Comm *, int *, ...);

The following are deprecated Fortran user-defined callback subroutine prototypes. The deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be declared like this:

SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR
LOGICAL FLAG

SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR

The deprecated handler-function for error handlers should be declared like this:

SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE, ........)
INTEGER COMM, ERROR_CODE

A.1.5 Info Keys

access_style
appnum
arch
cb_block_size
cb_buffer_size
cb_nodes
chunked_item
chunked_size
chunked
collective_buffering
file_perm
filename

typedef void Datarep_conversion_function(void* userbuf, Datatype& datatype, int count, void* filebuf, Offset position, void* extra_state);

A.1.4 Deprecated prototype definitions

The following are defined C typedefs for deprecated user-defined functions, also included in the file mpi.h.

/* prototypes for user-defined functions */
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval, void *extra_state, void *attribute_val_in, void *attribute_val_out, int *flag);
typedef int MPI_Delete_function(MPI_Comm comm, int keyval, void *attribute_val, void *extra_state);
typedef void MPI_Handler_function(MPI_Comm *, int *, ...);

The following are deprecated Fortran user-defined callback subroutine prototypes. The deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be declared like these:

SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERR
LOGICAL FLAG

SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR

The deprecated handler-function for error handlers should be declared like this:

SUBROUTINE HANDLER_FUNCTION(COMM, ERROR_CODE, ........)
INTEGER COMM, ERROR_CODE

A.1.5 Info Keys

access_style
appnum
arch
cb_block_size
cb_buffer_size
cb_nodes
chunked_item
chunked_size
chunked
collective_buffering
file_perm
filename
A.1. DEFINED VALUES AND HANDLES

file
host
io_node_list
ip_address
ip_port
nb_proc
no_locks
num_io_nodes
path
soft
striping_factor
striping_unit
wdir

A.1.6 Info Values

false
random
read_mostly
read_once
reverse_sequential
sequential
true
write_mostly
write_once
A.2 C Bindings

A.2.1 Point-to-Point Communication C Bindings

int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
int MPI_Buffer_attach(void* buffer, int size)
int MPI_Buffer_detach(void* buffer_addr, int* size)
int MPI_Cancel(MPI_Request *request)
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
    MPI_Status *status)
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Request_free(MPI_Request *request)
int MPI_Request_get_status(MPI_Request request, int *flag,
    MPI_Status *status)
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)

int MPI_Sendrecv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
int dest, int sendtag, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
MPI_Status *status)

int MPI_Send(void* buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)

int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)

int MPI_Startall(int count, MPI_Request *array_of_requests)

int MPI_Start(MPI_Request *request)

int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
MPI_Status *array_of_statuses)

int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
int *flag, MPI_Status *status)

int MPI_Test_cancelled(MPI_Status *status, int *flag)

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

int MPI_Testsome(int incount, MPI_Request *array_of_requests,
int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)

int MPI_Waitall(int count, MPI_Request *array_of_requests,
MPI_Status *array_of_statuses)

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
MPI_Status *status)

int MPI_Wait(MPI_Request *request, MPI_Status *status)

int MPI_Waitsome(int incount, MPI_Request *array_of_requests,
int *outcount, int *array_of_indices, MPI_Status *array_of_statuses)

A.2.2 Datatypes C Bindings

int MPI_Get_address(void *location, MPI_Aint *address)

int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)

int MPI_Pack_external(char *datarep, void *inbuf, int incount,
A.2 C BINDINGS

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, 
MPI_Aint *extent)
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, 
MPI_Aint *true_extent)
int MPI_Type_indexed(int count, int *array_of_blocklengths, 
int *array_of_displacements, MPI_Datatype oldtype, 
MPI_Datatype *newtype)
int MPI_Type_size(MPI_Datatype datatype, int *size)
int MPI_Type_vector(int count, int blocklength, int stride, 
MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize, 
MPI_Aint *position, void *outbuf, int outcount, 
MPI_Datatype datatype)
int MPI_Unpack(void* inbuf, int insize, int *position, void *outbuf, 
int outcount, MPI_Datatype datatype, MPI_Comm comm)

A.2.3 Collective Communication C Bindings

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, 
void* recvbuf, int recvcount, MPI_Datatype recvtype, 
MPI_Comm comm)
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, 
void* recvbuf, int *recvcounts, int *displs, 
MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, 
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, 
void* recvbuf, int recvcount, MPI_Datatype recvtype, 
MPI_Comm comm)
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, 
MPI_Datatype sendtypes[], void* recvbuf, int *recvcounts, 
int *rdispls, MPI_Datatype recvtypes[], MPI_Comm comm)
int MPI_Barrier(MPI_Comm comm)
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, 
MPI_Comm comm)
int MPI_Exscan(void *sendbuf, void *recvbuf, int count, 
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int *recvcounts, int *displs,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
int MPI_op_free(MPI_Op *op)
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int*recvcounts,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
    void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
    MPI_Datatype sendtype, void* recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)

A.2.4 Groups, Contexts, Communicators, and Caching C Bindings
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
    MPI_Comm_delete_attr_function *comm_delete_attr_fn,
    int *comm_keyval, void *extra_state)
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
int MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,
    void *attribute_val_in, void *attribute_val_out, int *flag)
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_free_keyval(int *comm_keyval)
int MPI_Comm_free(MPI_Comm *comm)
int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
    int *flag)
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
int MPI_COMM_NULL_COPY_FN(MPI_Comm oldcomm, int comm_keyval, 
    void *extra_state, void *attribute_val_in, 
    void *attribute_val_out, int *flag)

int MPI_COMM_NULL_DELETE_FN(MPI_Comm comm, int comm_keyval, void *
    attribute_val, void *extra_state)

int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)
int MPI_Comm_set_name(MPI_Comm comm, char *comm_name)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2, 
    MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_free(MPI_Group *group)
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, 
    MPI_Group *newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], 
    MPI_Group *newgroup)
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], 
    MPI_Group *newgroup)
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_size(MPI_Group group, int *size)
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1, 
    MPI_Group group2, int *ranks2)
int MPI_Group_union(MPI_Group group1, MPI_Group group2, 
    MPI_Group *newgroup)
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, 
    MPI_Comm peer_comm, int remote_leader, int tag, 
    MPI_Comm *newintercomm)
ANNEX A. LANGUAGE BINDINGS SUMMARY

int MPI_Intercomm_merge(MPI_Comm intercomm, int high, 
MPI_Comm *newintracomm)

int MPI_Type_create_keyval(MPI_Type_copy_attr_function *type_copy_attr_fn, 
MPI_Type_delete_attr_function *type_delete_attr_fn, 
int *type_keyval, void *extra_state)

int MPI_Type_delete_attr(MPI_Datatype type, int type_keyval)

int MPI_TYPE_DUP_FN(MPI_Datatype oldtype, int type_keyval, 
void *extra_state, void *attribute_val_in, 
void *attribute_val_out, int *flag)

int MPI_Type_free_keyval(int *type_keyval)

int MPI_Type_get_attr(MPI_Datatype type, int type_keyval, void 
*attribute_val, int *flag)

int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen)

int MPI_TYPE_NULL_COPY_FN(MPI_Datatype oldtype, int type_keyval, 
void *extra_state, void *attribute_val_in, 
void *attribute_val_out, int *flag)

int MPI_Type_set_attr(MPI_Datatype type, int type_keyval, 
void *attribute_val)

int MPI_Type_set_name(MPI_Datatype type, char *type_name)

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)

int MPI_Win_delete_attr(MPI_Win win, int win_keyval)

int MPI_WIN_DUP_FN(MPI_Win oldwin, int win_keyval, void *extra_state, 
void *attribute_val_in, void *attribute_val_out, int *flag)

int MPI_Win_free_keyval(int *win_keyval)

int MPI_Win_get_attr(MPI_Win win, int win_keyval, void *attribute_val, 
int *flag)

int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)

int MPI_WIN_NULL_COPY_FN(MPI_Win oldwin, int win_keyval, void *extra_state, 
void *attribute_val_in, void *attribute_val_out, int *flag)

int MPI_WIN_NULL_DELETE_FN(MPI_Win win, int win_keyval, void *extra_state)

int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)

int MPI_Win_set_name(MPI_Win win, char *win_name)
A.2.5 Process Topologies C Bindings

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods, int *newrank)
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
int MPI_Dims_create(int nnodes, int ndims, int *dims)
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder, MPI_Comm *comm_graph)
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges)
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges, int *newrank)
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)
int MPI_Topo_test(MPI_Comm comm, int *status)

A.2.6 MPI Environmenta Management C Bindings

int MPI_Abort(MPI_Comm comm, int errorcode)
int MPI_Add_error_class(int *errorclass)
int MPI_Add_error_code(int errorclass, int *errorcode)
int MPI_Add_error_string(int errorcode, char *string)
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn *function, MPI_Errhandler *errhandler)
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn *function, MPI_Errhandler *errhandler)
A.2.7 The Info Object C Bindings

int MPI_Info_create(MPI_Info *info)
int MPI_Info_delete(MPI_Info info, char *key)
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
int MPI_Info_free(MPI_Info *info)
int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value, 
int *flag)
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
int MPI_Info_get_valuelen(MPI_Info info, char *key, int *valuelen, int *flag)
int MPI_Info_set(MPI_Info info, char *key, char *value)

A.2.8 Process Creation and Management C Bindings
int MPI_Close_port(char *port_name)
int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_disconnect(MPI_Comm *comm)
int MPI_Comm_get_parent(MPI_Comm *parent)
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
int MPI_Comm_spawn_multiple(int count, char *array_of_commands[], char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[], int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)
int MPI_Open_port(MPI_Info info, char *port_name)
int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)
int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)

A.2.9 One-Sided Communications C Bindings
int MPI_Accumulate(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_Get(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_Win win)
int MPI_Put(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_Win win)
int MPI_Win_complete(MPI_Win win)
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, MPI_Win *win)
int MPI_Win_fence(int assert, MPI_Win win)
int MPI_Win_free(MPI_Win *win)
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_test(MPI_Win win, int *flag)
int MPI_Win_unlock(int rank, MPI_Win win)
int MPI_Win_wait(MPI_Win win)

A.2.10 External Interfaces C Bindings
int MPI_Grequest_complete(MPI_Request request)
int MPI_Grequest_start(MPI_Grequest_query_function *query_fn,
MPI_Grequest_free_function *free_fn,
MPI_Grequest_cancel_function *cancel_fn, void *extra_state,
MPI_Request *request)
int MPI_Init_thread(int *argc, char *((*argv)[0]), int required,
int *provided)
int MPI_Is_thread_main(int *flag)
int MPI_Query_thread(int *provided)
int MPI_Status_set_cancelled(MPI_Status *status, int flag)
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype,
int count)

A.2.11 I/O C Bindings
int MPI_File_close(MPI_File *fh)
int MPI_File_delete(char *filename, MPI_Info info)
int MPI_File_get_amode(MPI_File fh, int *amode)
int MPI_File_get_atomicity(MPI_File fh, int *flag)
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
MPI_Offset *disp)
int MPI_File_close(MPI_File *fh)
int MPI_File_delete(char *filename, MPI_Info info)
int MPI_File_get_amode(MPI_File fh, int *amode)
int MPI_File_get_atomicity(MPI_File fh, int *flag)
int MPI_File_get_byte_offset(MPI_File fh, MPI_Offset offset,
MPI_Offset *disp)
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
MPI_Aint *extent)
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
MPI_Datatype *filetype, char *datarep)
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iread(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iread_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info,
MPI_File *fh)
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_read_all(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype)
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype, MPI_Status *status)
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(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_read_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_read_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
int MPI_File_seek_shared(MPI_File fh, MPI_Offset offset, int whence)
int MPI_File_set_atomicity(MPI_File fh, int flag)
int MPI_File_set_info(MPI_File fh, MPI_Info info)
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
int MPI_File_sync(MPI_File fh)
int MPI_File_write_all_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)
int MPI_File_write_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype)
int MPI_File_write_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
int MPI_File_write_ordered_begin(MPI_File fh, void *buf, int count, MPI_Datatype datatype)
int MPI_File_write_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
int MPI_File_write_ordered(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
A.2. C BINDINGS

int MPI_File_write_shared(MPI_File fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status)

int MPI_Register_datarep(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)

A.2.12 Language Bindings C Bindings

int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *type)

MPI_Fint MPI_Comm_c2f(MPI_Comm comm) MPI_Comm MPI_Comm_f2c(MPI_Fint comm)
MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler) MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
MPI_Fint MPI_File_c2f(MPI_File file) MPI_File MPI_File_f2c(MPI_Fint file)
MPI_Fint MPI_Group_c2f(MPI_Group group) MPI_Group MPI_Group_f2c(MPI_Fint group)
MPI_Fint MPI_Info_c2f(MPI_Info info) MPI_Info MPI_Info_f2c(MPI_Fint info)
MPI_Fint MPI_Op_c2f(MPI_Op op) MPI_Op MPI_Op_f2c(MPI_Fint op)
MPI_Fint MPI_Request_c2f(MPI_Request request) MPI_Request MPI_Request_f2c(MPI_Fint request)
int MPI_Status_c2f(MPI_Status *status, MPI_Fint *f_status)
int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype) MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Win_c2f(MPI_Win win) MPI_Win MPI_Win_f2c(MPI_Fint win)
A.2.13 Profiling Interface C Bindings

int MPI_Pcontrol(const int level, ...)

A.2.14 Deprecated C Bindings

int MPI_Address(void* location, MPI_Aint *address)
int MPI_Attr_delete(MPI_Comm comm, int keyval)
int MPI_Attr_get(MPI_Comm comm, int keyval, void *attribute_val, int *flag)
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
int MPI_DUP_FN(MPI_Comm oldcomm, int keyval, void* extra_state,
    void *attribute_val_in, void *attribute_val_out, int *flag)
int MPI_Errhandler_create(MPI_Handler_function *function,
    MPI_Errhandler *errhandler)
int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function
    *delete_fn, int *keyval, void* extra_state)
int MPI_Keyval_free(int *keyval)
int MPI_NULL_COPY_FN(MPI_Comm oldcomm, int keyval, void* extra_state,
    void *attribute_val_in, void *attribute_val_out, int *flag)
int MPI_NULL_DELETE_FN(MPI_Comm comm, int keyval, void *attribute_val,
    void *extra_state)
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
    MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
int MPI_Type_struct(int count, int *array_of_blocklengths,
    MPI_Aint *array_of_displacements,
    MPI_Datatype *array_of_types, MPI_Datatype *newtype)
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
A.3 Fortran Bindings

A.3.1 Point-to-Point Communication Fortran Bindings

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  <type> BUF(*) INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
  <type> BUFFER(*) INTEGER SIZE, IERROR

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
  <type> BUFFER_ADDR(*) INTEGER SIZE, IERROR

MPI_CANCEL(REQUEST, IERROR)
  INTEGER REQUEST, IERROR

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
  INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

MPI_IBSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
  LOGICAL FLAG INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
  <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  <type> BUF(*) INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
  INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
  <type> BUF(*) INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR
ANNEX A. LANGUAGE BINDINGS SUMMARY

1   INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
2   MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
3   INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
4   MPI_REQUEST_FREE(REQUEST, IERROR)
5   INTEGER REQUEST, IERROR
6   MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
7   INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
8   LOGICAL FLAG
9   MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
10  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
11  MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
12  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
13  MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
14  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
15  MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
16  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
17  MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)
18  INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR
19  MPI_SENDRECV(SENDBUF, SENDCOUNT, SENTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR)
20  INTEGER SENDBUF(*), RECVBUF(*)
21  MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
22  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
23  MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
24  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
25  MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
26  INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
27  MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
28  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
29  MPI_REQUEST_FREE(REQUEST, IERROR)
30  INTEGER REQUEST, IERROR
31  MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
32  INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
33  LOGICAL FLAG
34  MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
35  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
36  MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
37  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
38  MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
39  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
40  MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
41  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
42  MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)
43  INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR
44  MPI_SENDRECV(SENDBUF, SENDCOUNT, SENTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR)
45  INTEGER SENDBUF(*), RECVBUF(*)
46  MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
47  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
48  MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
49  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
50  MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
51  INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
A.3. FORTRAN Bindings

```
MPI_START(Request, IERROR)
   INTEGER Request, IERROR

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
   INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

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

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

MPI_TEST(Request, FLAG, STATUS, IERROR)
   LOGICAL FLAG
   INTEGER Request, STATUS(MPI_STATUS_SIZE), IERROR

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 
   ARRAY_OF_STATUSES, IERROR)
   INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 
   ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
   INTEGER COUNT, ARRAY_OF_REQUESTS(*) INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
   INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR

MPI_WAIT(Request, STATUS, IERROR)
   INTEGER Request, STATUS(MPI_STATUS_SIZE), IERROR

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 
   ARRAY_OF_STATUSES, IERROR)
   INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 
   ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

A.3.2 Datatypes Fortran Bindings

MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
   INTEGER LOCATION(*) INTEGER IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS

MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
   INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

A.3. FORTRAN Bindings

```
MPI_START(Request, IERROR)
   INTEGER Request, IERROR

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
   LOGICAL FLAG
   INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

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

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

MPI_TEST(Request, FLAG, STATUS, IERROR)
   LOGICAL FLAG
   INTEGER Request, STATUS(MPI_STATUS_SIZE), IERROR

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 
   ARRAY_OF_STATUSES, IERROR)
   INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 
   ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)
   INTEGER COUNT, ARRAY_OF_REQUESTS(*) INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
   INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR

MPI_WAIT(Request, STATUS, IERROR)
   INTEGER Request, STATUS(MPI_STATUS_SIZE), IERROR

MPI_WAITSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, 
   ARRAY_OF_STATUSES, IERROR)
   INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT, ARRAY_OF_INDICES(*), 
   ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

A.3.2 Datatypes Fortran Bindings

MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
   INTEGER LOCATION(*) INTEGER IERROR INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS

MPI_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
   INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```
ANNEX A. LANGUAGE BINDINGS SUMMARY

MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, IERROR)
INTEGER INCOUNT, DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
CHARACTER(*) DATAREP

MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
CHARACTER(*) DATAREP

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR

MPI_TYPE_COMMIT(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES, Array_OF_DISTRIBUTIONS, Array_OF_PSIZES, ORDER, OLDTYPE, NEWTYPE, IERROR)
INTEGER SIZE, RANK, NDIMS, Array_OF_GSIZES(*), Array_OF_DISTRIBUTIONS(*), Array_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_CREATE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS, Array_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), Array_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
INTEGER OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT

MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, Array_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), Array_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR
A.3. FORTRAN BINDINGS

 INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)

 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

 MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)
 INTEGER TYPE, NEWTYPE, IERROR

 MPI_TYPE_FREE(DATATYPE, IERROR)
 INTEGER DATATYPE, IERROR

 MPI_TYPE_GET_CONTENTS(DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES, 
 ARRAY_OF_INTEGERS, ARRAY_OF_ADDRESSES, ARRAY_OF_DATATYPES, 
 IERROR)
 INTEGER DATATYPE, MAX_INTEGERS, MAX_ADDRESSES, MAX_DATATYPES, 
 ARRAY_OF_INTEGERS(*), ARRAY_OF_ADDRESSES(*), ARRAY_OF_DATATYPES(*), IERROR

 MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, 
 COMBINER, IERROR)
 INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR

 MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
 INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT

 MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
 INTEGER DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) TRUE_LB, TRUE_EXTENT

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

 MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
 INTEGER DATATYPE, SIZE, IERROR

 MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
 INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

 MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, 
 DATATYPE, IERROR)
 INTEGER OUTCOUNT, DATATYPE, IERROR INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
 CHARACTER(*) DATAREP <type> INBUF(*), OUTBUF(*)

 MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
A.3.3 Collective Communication Fortran Bindings

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVCYPE, COMM, IERROR)

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)

MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)

MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)

MPI_BARRIER(COMM, IERROR)

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)

MPI_HALLGATHER(SENDUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVCYPE, COMM, IERROR)

MPI_ALLGATHERV(SENDUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)

MPI_ALLREDUCE(SENDUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)

MPI_ALLTOALL(SENDUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)

MPI_ALLTOALLV(SENDUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)

MPI_ALLTOALLW(SENDUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)

MPI_BARRIER(COMM, IERROR)

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

MPI_EXSCAN(SENDFU, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)

MPI_GATHER(SENDUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROU, COMM, IERROR)
A.3. FORTRAN BINDINGS

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT, COMM, IERROR

MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
EXTERNAL FUNCTIONLOGICAL COMMUTEINTEGER OP, IERROR
MPI_OP_FREE( OP, IERROR)
INTEGER OP, IERROR
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*) INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*) INTEGER COUNT, DATATYPE, OP, COMM, IERROR

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*) INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
A.3.4 Groups, Contexts, Communicators, and Caching Fortran Bindings

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
INTEGER COMM1, COMM2, RESULT, IERROR

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
INTEGER COMM, GROUP, NEWCOMM, IERROR

MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL, EXTRA_STATE, IERROR)
EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN INTEGER COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
INTEGER COMM, COMM_KEYVAL, IERROR
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
INTEGER OLDCOMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT
LOGICAL FLAG
MPI_COMM_FREE(COMM, IERROR)
INTEGER COMM, IERROR
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
INTEGER COMM_KEYVAL, IERROR
MPI_COMM_GET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
INTEGER COMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
LOGICAL FLAG
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
INTEGER COMM, RESULTLEN, IERROR
CHARACTER(*) COMM_NAME
MPI_COMM_GROUP(COMM, GROUP, IERROR)
INTEGER COMM, GROUP, IERROR
MPI_COMM_NULL_COPY_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
INTEGER OLDCOMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT
LOGICAL FLAG
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
A.3. FORTRAN BINDINGS

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_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_TEST_INTER(COMM, FLAG, IERROR)
  INTEGER COMM, IERROR
  LOGICAL FLAG

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
  INTEGER GROUP1, GROUP2, RESULT, IERROR

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
  INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
  INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_FREE(GROUP, IERROR)
  INTEGER GROUP, IERROR

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
  INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
  INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
  INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
  INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_GROUP_RANK(GROUP, RANK, IERROR)
  INTEGER GROUP, RANK, IERROR

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
  INTEGER GROUP, SIZE, IERROR

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
  INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
  INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
ANNEX A. LANGUAGE BINDINGS SUMMARY

INTEGER TAG, NEWINTERCOMM, IERROR
INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
NEWINTERCOMM, IERROR

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)
INTEGER INTERCOMM, INTRACOMM, IERROR
LOGICAL HIGH

MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL,
EXTRA_STATE, IERROR)
EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
INTEGER TYPE_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_TYPE_DELETE_ATTR(TYPE, TYPE_KEYVAL, IERROR)
INTEGER TYPE, TYPE_KEYVAL, IERROR

MPI_TYPE_DUP_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT
LOGICAL FLAG

MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
INTEGER TYPE_KEYVAL, IERROR

MPI_TYPE_GET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
INTEGER TYPE, TYPE_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
LOGICAL FLAG

MPI_TYPE_GET_NAME(TYPE, TYPE_NAME, RESULTLEN, IERROR)
INTEGER TYPE, RESULTLEN, IERROR
CHARACTER(*) TYPE_NAME

MPI_TYPE_NULL_COPY_FN(OLDTYPE, TYPE_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERROR)
INTEGER OLDTYPE, TYPE_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT
LOGICAL FLAG

MPI_TYPE_NULL_DELETE_FN(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
IERROR)
INTEGER TYPE, TYPE_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

MPI_TYPE_SET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
INTEGER TYPE, TYPE_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_TYPE_SET_NAME(TYPE, TYPE_NAME, IERROR)
A.3. FORTRAN BINDINGS

INTEGER TYPE, IERROR
CHARACTER(*) TYPE_NAME

MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL, EXTRA_STATE, IERROR)
EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
INTEGER WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
INTEGER WIN, WIN_KEYVAL, IERROR

MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
INTEGER OLDWIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
LOGICAL FLAG

MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
INTEGER WIN_KEYVAL, IERROR

MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
INTEGER WIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
LOGICAL FLAG

MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
INTEGER WIN, RESULTLEN, IERROR
CHARACTER(*) WIN_NAME

MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
INTEGER OLDWIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT
LOGICAL FLAG

MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
INTEGER WIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
INTEGER WIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
INTEGER WIN, IERROR
CHARACTER(*) WIN_NAME
ANNEX A. LANGUAGE BINDINGS SUMMARY

A.3.5 Process Topologies Fortran Bindings

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS(*), IERROR)
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
   LOGICAL PERIODS(*), REORDER

MPI_CART_DIM_GET(COMM, NDIMS, IERROR)
INTEGER COMM, NDIMS, IERROR

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
   LOGICAL PERIODS(*)

MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
   LOGICAL PERIODS(*)

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
INTEGER COMM, COORDS(*), RANK, IERROR

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
   LOGICAL REMAIN_DIMS(*)

MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
INTEGER NNODES, NDIMS, DIMS(*), IERROR

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, IERROR)
INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
   LOGICAL REORDER

MPI_GRAPH_DIMS_GET(COMM, NNODES, NEDGES, IERROR)
INTEGER COMM, NNODES, NEDGES, IERROR

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR

MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
INTEGER COMM, RANK, NNEIGHBORS, IERROR

MPI_TOPO_TEST(COMM, STATUS, IERROR)
INTEGER COMM, STATUS, IERROR
A.3. FORTRAN BINDINGS

A.3.6 MPI Environment Management Fortran Bindings

DOUBLE PRECISION MPI_WTICK()
DOUBLE PRECISION MPI_WTIME()

MPI_ABORT(COMM, ERRORCODE, IERROR)
  INTEGER COMM, ERRORCODE, IERROR

MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
  INTEGER ERRORCLASS, IERROR

MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
  INTEGER ERRORCLASS, ERRORCODE, IERROR

MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
  INTEGER ERRORCODE, IERROR
    CHARACTER(*) STRING

MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
  INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR

MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
  INTEGER COMM, ERRORCODE, IERROR

MPI_COMM_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
  EXTERNAL FUNCTION INTEGER ERRHANDLER, IERROR

MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
  INTEGER COMM, ERRHANDLER, IERROR

MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
  INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
  INTEGER ERRHANDLER, IERROR

MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
  INTEGER ERRORCODE, ERRORCLASS, IERROR

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
  INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER(*) STRING

MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
  INTEGER FH, ERRORCODE, IERROR

MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
  EXTERNAL FUNCTION INTEGER ERRHANDLER, IERROR

MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
  INTEGER FILE, ERRHANDLER, IERROR
ANNEX A. LANGUAGE BINDINGS SUMMARY

A.3.7 The Info Object Fortran Bindings

- **MPI_INFO_CREATE(INFO, IERROR)**
  - INTEGER INFO, IERROR

- **MPI_INFO_DELETE(INFO, KEY, IERROR)**
  - INTEGER INFO, IERROR
  - CHARACTER*(*) KEY

- **MPI_INFO_DUP(INFO, NEWINFO, IERROR)**
  - INTEGER INFO, NEWINFO, IERROR

- **MPI_INFO_FREE(INFO, IERROR)**
  - INTEGER INFO, IERROR
A.3. FORTRAN BINDINGS

INTEGER INFO, IERROR

MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
INTEGER INFO, VALUELEN, IERROR
CHARACTER(*) KEY, VALUE
LOGICAL FLAG

MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
INTEGER INFO, NKEYS, IERROR

MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
INTEGER INFO, N, IERROR
CHARACTER(*) KEY

MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
INTEGER INFO, VALUELEN, IERROR
LOGICAL FLAG
CHARACTER(*) KEY

MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
INTEGER INFO, IERROR
CHARACTER(*) KEY, VALUE

A.3.8 Process Creation and Management Fortran Bindings

MPI_CLOSE_PORT(PORT_NAME, IERROR)
CHARACTER(*) PORT_NAME
INTEGER IERROR

MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
CHARACTER(*) PORT_NAME
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
CHARACTER(*) PORT_NAME
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

MPI_COMM_DISCONNECT(COMM, IERROR)
INTEGER COMM, IERROR

MPI_COMM_GET_PARENT(PARENT, IERROR)
INTEGER PARENT, IERROR

MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
INTEGER FD, INTERCOMM, IERROR

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES, IERROR)
CHARACTER(*) COMMAND, ARGV(*)
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR

MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
ARRAY_OF_ERRCODES, IERROR)
INTEGER COUNT, COMMAND(*)
CHARACTER(*) ARGV(*)
INTEGER ARRAY_OF_COMMANDS, ARRAY_OF_ARGV, ARRAY_OF_ERRCODES(*)

A.3. FORTRAN BINDINGS

INTEGER INFO, IERROR

MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
INTEGER INFO, VALUELEN, IERROR
CHARACTER(*) KEY, VALUE
LOGICAL FLAG

MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
INTEGER INFO, NKEYS, IERROR

MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
INTEGER INFO, N, IERROR
CHARACTER(*) KEY

MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
INTEGER INFO, VALUELEN, IERROR
LOGICAL FLAG
CHARACTER(*) KEY

MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
INTEGER INFO, IERROR
CHARACTER(*) KEY, VALUE

A.3.8 Process Creation and Management Fortran Bindings

MPI_CLOSE_PORT(PORT_NAME, IERROR)
CHARACTER(*) PORT_NAME
INTEGER IERROR

MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
CHARACTER(*) PORT_NAME
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
CHARACTER(*) PORT_NAME
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR

MPI_COMM_DISCONNECT(COMM, IERROR)
INTEGER COMM, IERROR

MPI_COMM_GET_PARENT(PARENT, IERROR)
INTEGER PARENT, IERROR

MPI_COMM_JOIN(FD, INTERCOMM, IERROR)
INTEGER FD, INTERCOMM, IERROR

MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES, IERROR)
CHARACTER(*) COMMAND, ARGV(*)
INTEGER INFO, MAXPROCS, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR

MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
ARRAY_OF_ERRCODES, IERROR)
INTEGER COUNT, COMMAND(*)
CHARACTER(*) ARGV(*)
INTEGER ARRAY_OF_COMMANDS, ARRAY_OF_ARGV, ARRAY_OF_ERRCODES(*)
A.3.9 One-Sided Communications Fortran Bindings

MPI_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 
    TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, OP, WIN, IERROR

MPI_GET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 
    TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR

MPI_PUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, 
    TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
    TARGET_DATATYPE, WIN, IERROR

MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
A.3. FORTRAN BINDINGS

INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR

MPI_WIN_FENCE(ASSERT, WIN, IERROR)
INTEGER ASSERT, WIN, IERROR

MPI_WIN_FREE(WIN, IERROR)
INTEGER WIN, IERROR

MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
INTEGER WIN, GROUP, IERROR

MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR

MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
INTEGER GROUP, ASSERT, WIN, IERROR

MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
INTEGER GROUP, ASSERT, WIN, IERROR

MPI_WIN_TEST(WIN, FLAG, IERROR)
INTEGER WIN, IERRORLOGICAL FLAG

MPI_WIN_UNLOCK(RANK, WIN, IERROR)
INTEGER RANK, WIN, IERROR

MPI_WIN_WAIT(WIN, IERROR)
INTEGER WIN, IERROR

A.3.10 External Interfaces Fortran Bindings

MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
INTEGER REQUEST, IERROR

MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST, IERROR)
INTEGER REQUEST, IERROREXTERNAL QUERY_FN, FREE_FN, CANCEL_FNINTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
INTEGER REQUIRED, PROVIDED, IERROR

MPI_IS_THREAD_MAIN(FLAG, IERROR)
INTEGER FLAG, IERROR

MPI_QUERY_THREAD(PROVIDED, IERROR)
INTEGER PROVIDED, IERROR

MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), IERRORLOGICAL FLAG
ANNEX A. LANGUAGE BINDINGS SUMMARY

MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

A.3.11 I/O Fortran Bindings

MPI_FILE_CLOSE(FH, IERROR)
INTEGER FH, IERROR

MPI_FILE_DELETE(FILENAME, INFO, IERROR)
CHARACTER*(*) FILENAME
INTEGER INFO, IERROR

MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
INTEGER FH, AMODE, IERROR

MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
INTEGER FH, IERROR
LOGICAL FLAG

MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP

MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
INTEGER FH, GROUP, IERROR

MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
INTEGER FH, INFO_USED, IERROR

MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE

MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
INTEGER FH, DATATYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT

MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
INTEGER FH, ETYPE, FILETYPE, IERROR
CHARACTER*(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP

MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
A.3. FORTRAN BINDINGS

INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_IREAD(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_FILE_IREAD_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_FILE_IWRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_IWRITE(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_FILE_IWRITE_SHARED(FH, BUF, COUNT, DATATYPE, REQUEST, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR

MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
  CHARACTER(*) FILENAME INTEGER COMM, AMODE, INFO, FH, IERROR

MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
  INTEGER FH, IERROR INTEGER(KIND=MPI_OFFSET_KIND) SIZE

MPI_FILE_READ_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
  INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
  INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_READ_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, IERROR
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
<type> BUF(*)
INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
INTEGER FH, WHENCE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
INTEGER FH, WHENCE, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
INTEGER FH, IERROR
LOGICAL FLAG
MPI_FILE_SET_INFO(FH, INFO, IERROR)
INTEGER FH, INFO, IERROR
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO, IERROR)
INTEGER FH, ETYPE, FILETYPE, INFO, IERROR
CHARACTER(*) DATAREP
INTEGER(KIND=MPI_OFFSET_KIND) DISP
MPI_FILE_SYNC(FH, IERROR)
INTEGER FH, IERROR
A.3. FORTRAN BINDINGS

MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)

MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_WRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, IERROR

MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
A.3.12 Language Bindings Fortran Bindings

MPI_SIZEOF(X, SIZE, IERROR)
   <type> X INTEGER SIZE, IERROR

MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
   INTEGER P, R, NEWTYPE, IERROR

MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
   INTEGER R, NEWTYPE, IERROR

MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
   INTEGER P, R, NEWTYPE, IERROR

MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, TYPE, IERROR)
   INTEGER TYPECLASS, SIZE, TYPE, IERROR

A.3.13 Profiling Interface Fortran Bindings

MPI_PCONTROL(LEVEL)
   INTEGER LEVEL, ...

A.3.14 Deprecated Fortran Bindings

MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
   <type> LOCATION(*) INTEGER ADDRESS, IERROR

MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
   INTEGER COMM, KEYVAL, IERROR

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
   INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
   LOGICAL FLAG

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
   INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERROR)
   INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, IERROR
   LOGICAL FLAG

MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)
A.3. FORTRAN BINDINGS

EXTERNAL FUNCTION INTEGER ERRHANDLER, IERROR

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
EXTERNAL COPY_FN, DELETE_FN INTEGER KEYVAL, EXTRA_STATE, IERROR

MPI_KEYVAL_FREE(KEYVAL, IERROR)
INTEGER KEYVAL, IERROR

MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG

MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR

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

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
ARRAY_OF_TYPES(*), NEWTYPE, IERROR

MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG

SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)

EXTERNAL FUNCTION INTEGER ERRHANDLER, IERROR

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
EXTERNAL COPY_FN, DELETE_FN INTEGER KEYVAL, EXTRA_STATE, IERROR

MPI_KEYVAL_FREE(KEYVAL, IERROR)
INTEGER KEYVAL, IERROR

MPI_NULL_COPY_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG

MPI_NULL_DELETE_FN(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR

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

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
ARRAY_OF_TYPES(*), NEWTYPE, IERROR

MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, FLAG, IERR)
INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
ATTRIBUTE_VAL_OUT, IERR LOGICAL FLAG

SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR

INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
A.4 C++ Bindings

A.4.1 Point-to-Point Communication C++ Bindings

namespace MPI {
    void Attach_buffer(void* buffer, int size)
    void Comm::Bsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
    Preaccept Comm::Bsend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
    void Request::Cancel() const
    int Detach_buffer(void* buffer)
    void Request::Free()
    int Status::Get_count(const Datatype& datatype) const
    int Status::Get_error() const
    int Status::Get_source() const
    bool Request::Get_status() const
    bool Request::Get_status(Status& status) const
    int Status::Get_tag() const
    Request Comm::Ibsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
    bool Comm::Iprobe(int source, int tag) const
    bool Comm::Iprobe(int source, int tag, Status& status) const
    Request Comm::Irecv(void* buf, int count, const Datatype& datatype, int source, int tag) const
    Request Comm::Irsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
    bool Status::Is_cancelled() const
    Request Comm::Isend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
    Request Comm::Issend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
    void Comm::Probe(int source, int tag) const
    void Comm::Probe(int source, int tag, Status& status) const
    Preaccept Comm::Recv_init(void* buf, int count, const Datatype& datatype, int source, int tag) const
}
ANNEX A. LANGUAGE BINDINGS SUMMARY

```c
void Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag) const
void Comm::Recv(void* buf, int count, const Datatype& datatype, int source, int tag, Status& status) const
void Comm::Rsend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
Prequest Comm::Rsend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
void Comm::Send(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
Prequest Comm::Send_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
void Comm::Sendrecv(const void* sendbuf, int sendcount, const Datatype& sendtype, int dest, int sendtag, void* recvbuf, int recvcount, const Datatype& recvtype, int source, int recvtag) const
void Comm::Sendrecv_replace(void* buf, int count, const Datatype& datatype, int dest, int sendtag, int source, int recvtag) const
void Comm::Ssend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
Prequest Comm::Ssend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
void Comm::Ssend(const void* buf, int count, const Datatype& datatype, int dest, int tag, Status& status) const
void Status::Set_error(int error)
void Status::Set_source(int source)
void Status::Set_tag(int tag)
void Comm::Ssend(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
Prequest Comm::Ssend_init(const void* buf, int count, const Datatype& datatype, int dest, int tag) const
static void Prequest::Startall(int count, Prequest array_of_requests[])
void Prequest::Start()
static bool Request::Testall(int count, Request array_of_requests[], Status array_of_statuses[])
static bool Request::Testall(int count, Request array_of_requests[], Status array_of_statuses[])
```
static bool Request::Testany(int count, Request array_of_requests[], int& index, Status& status)
static bool Request::Testany(int count, Request array_of_requests[], int& index)
bool Request::Test()
bool Request::Test(Status& status)
static int Request::Testsome(int incount, Request array_of_requests[], int array_of_indices[], Status array_of_statuses[])
static int Request::Testsome(int incount, Request array_of_requests[], int array_of_indices[])
static void Request::Waitall(int count, Request array_of_requests[], Status array_of_statuses[])
static void Request::Waitall(int count, Request array_of_requests[])
static int Request::Waitany(int count, Request array_of_requests[], Status& status)
static int Request::Waitany(int count, Request array_of_requests[])
void Request::Wait(Status& status)
static int Request::Waitsome(int incount, Request array_of_requests[], int array_of_indices[], Status array_of_statuses[])
static int Request::Waitsome(int incount, Request array_of_requests[], int array_of_indices[])
void Request::Wait();

A.4.2 Datatypes C++ Bindings
namespace MPI {
    void Datatype::Commit()
    Datatype Datatype::Create_contiguous(int count) const
    Datatype Datatype::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[],
        int order) const
    Datatype Datatype::Create_hindexed(int count,
        const int array_of_blocklengths[], const Aint array_of_displacements[])
};
Datatype Datatype::Create_hvector(int count, int blocklength, Aint stride) const
Datatype Datatype::Create_indexed_block(int count, int blocklength, const int array_of_displacements[]) const
Datatype Datatype::Create_indexed(int count, const int array_of_blocklengths[], const int array_of_displacements[]) const
Datatype Datatype::Create_resized(const Aint lb, const Aint extent) const
static Datatype Datatype::Create_struct(int count, const int array_of_blocklengths[], const Aint array_of_displacements[], const Datatype array_of_types[]) const
Datatype Datatype::Create_subarray(int ndims, const int array_of_sizes[], const int array_of_subsizes[], const int array_of_starts[], int order) const
Datatype Datatype::Create_vector(int count, int blocklength, int stride) const
Datatype Datatype::Dup() const
void Datatype::Free() const
Aint Get_address(void* location) const
void Datatype::Get_contents(int max_integers, int max_addresses, int max_datatypes, int array_of_integers[], Aint array_of_addresses[], Datatype array_of_datatypes[]) const
int Status::Get_elements(const Datatype& datatype) const
void Datatype::Get_envelope(int& num_integers, int& num_addresses, int& num_datatypes, int& combiner) const
void Datatype::Get_extent(Aint& lb, Aint& extent) const
int Datatype::Get_size() const
void Datatype::Get_true_extent(Aint& true_lb, Aint& true_extent) const
void Datatype::Pack(const void* inbuf, int incount, void* outbuf, int outsize, int& position, const Comm& comm) const
void Datatype::Pack_external(const char* datarep, const void* inbuf, int incount, void* outbuf, Aint outsize, Aint& position) const
Aint Datatype::Pack_external_size(const char* datarep, int incount) const
int Datatype::Pack_size(int incount, const Comm& comm) const
void Datatype::Unpack(const void* inbuf, int insize, void* outbuf, int outcount, int position, const Comm& comm) const
Datatype Datatype::Create_hvector(int count, int blocklength, Aint stride) const
Datatype Datatype::Create_indexed_block(int count, int blocklength, const int array_of_displacements[]) const
Datatype Datatype::Create_indexed(int count, const int array_of_blocklengths[], const int array_of_displacements[]) const
Datatype Datatype::Create_resized(const Aint lb, const Aint extent) const
static Datatype Datatype::Create_struct(int count, const int array_of_blocklengths[], const Aint array_of_displacements[], const Datatype array_of_types[]) const
Datatype Datatype::Create_subarray(int ndims, const int array_of_sizes[], const int array_of_subsizes[], const int array_of_starts[], int order) const
Datatype Datatype::Create_vector(int count, int blocklength, int stride) const
Datatype Datatype::Dup() const
void Datatype::Free() const
Aint Get_address(void* location)
void Datatype::Get_contents(int max_integers, int max_addresses, int max_datatypes, int array_of_integers[], Aint array_of_addresses[], Datatype array_of_datatypes[]) const
int Status::Get_elements(const Datatype& datatype) const
void Datatype::Get_envelope(int& num_integers, int& num_addresses, int& num_datatypes, int& combiner) const
void Datatype::Get_extent(Aint& lb, Aint& extent) const
int Datatype::Get_size() const
void Datatype::Get_true_extent(Aint& true_lb, Aint& true_extent) const
void Datatype::Pack(const void* inbuf, int incount, void* outbuf, int outsize, int position, const Comm& comm) const
void Datatype::Pack_external(const char* datarep, const void* inbuf, int incount, void* outbuf, Aint outsize, Aint& position) const
Aint Datatype::Pack_external_size(const char* datarep, int incount) const
int Datatype::Pack_size(int incount, const Comm& comm) const
void Datatype::Unpack(const void* inbuf, int insize, void* outbuf, int outcount, int position, const Comm& comm) const
A.4. C++ BINDINGS

void Datatype::Unpack_external(const char* datarep, const void* inbuf,
    Aint insize, Aint& position, void* outbuf, int outcount) const

A.4.3 Collective Communication C++ Bindings

namespace MPI {

void Comm::Allgather(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype) const = 0

void Comm::Allgatherv(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype) const = 0

void Comm::Allreduce(const void* sendbuf, void* recvbuf, int count, const
    Datatype& datatype, const Op& op) const = 0

void Comm::Alltoall(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype) const = 0

void Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
    const int sdispls[], const Datatype& sendtype, void* recvbuf,
    const int recvcounts[], const int rdispls[], const Datatype
    & recvtype) const = 0

void Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
    const int sdispls[], const Datatype sendtypes[], void* recvbuf,
    const int recvcounts[], const int rdispls[], const Datatype
    recvtypes[]) const = 0

void Comm::Barrier() const = 0

void Comm::Bcast(void* buffer, int count, const Datatype& datatype,
    int root) const = 0

void Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count, const
    Datatype& datatype, const Op& op) const

void Op::Free()  

void Comm::Gather(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype, int root) const = 0

void Comm::Gatherv(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    int displs[], const Datatype& recvtype, int root) const = 0

void Comm::Gather(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype, int root) const = 0

};

A.4. C++ BINDINGS

void Datatype::Unpack_external(const char* datarep, const void* inbuf,
    Aint insize, Aint& position, void* outbuf, int outcount) const

A.4.3 Collective Communication C++ Bindings

namespace MPI {

void Comm::Allgather(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype) const = 0

void Comm::Allgatherv(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, const int recvcounts[],
    const int displs[], const Datatype& recvtype) const = 0

void Comm::Allreduce(const void* sendbuf, void* recvbuf, int count, const
    Datatype& datatype, const Op& op) const = 0

void Comm::Alltoall(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype) const = 0

void Comm::Alltoallv(const void* sendbuf, const int sendcounts[],
    const int sdispls[], const Datatype& sendtype, void* recvbuf,
    const int recvcounts[], const int rdispls[], const Datatype
    & recvtype) const = 0

void Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
    const int sdispls[], const Datatype sendtypes[], void* recvbuf,
    const int recvcounts[], const int rdispls[], const Datatype
    recvtypes[]) const = 0

void Comm::Barrier() const = 0

void Comm::Bcast(void* buffer, int count, const Datatype& datatype,
    int root) const = 0

void Intracomm::Exscan(const void* sendbuf, void* recvbuf, int count, const
    Datatype& datatype, const Op& op) const

void Op::Free()  

void Comm::Gather(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    Datatype& recvtype, int root) const = 0

void Comm::Gatherv(const void* sendbuf, int sendcount, const
    Datatype& sendtype, void* recvbuf, int recvcount, const
    int displs[], const Datatype& recvtype, int root) const = 0

};
void Op::Init(User_function* function, bool commute)
void Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
const Datatype& datatype, const Op& op, int root) const = 0
void Comm::Reduce_scatter(const void* sendbuf, void* recvbuf,
int recvcounts[], const Datatype& datatype, const Op& op) const = 0
void Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
const Datatype& datatype, const Op& op) const
void Comm::Scatter(const void* sendbuf, int sendcount, const
Datatype& sendtype, void* recvbuf, int recvcount,
const Datatype& recvtype, int root) const = 0
void Comm::Scatterv(const void* sendbuf, const int sendcounts[],
const int displs[], const Datatype& sendtype, void* recvbuf,
int recvcount, const Datatype& recvtype, int root) const = 0

A.4.4 Groups, Contexts, Communicators, and Caching C++ Bindings
namespace MPI {
Comm& Comm::Clone() const = 0
Cartcomm& Cartcomm::Clone() const
Graphcomm& Graphcomm::Clone() const
Intercomm& Intercomm::Clone() const
Intracomm& Intracomm::Clone() const
static int Comm::Compare(const Comm& comm1, const Comm& comm2)
static int Group::Compare(const Group& group1, const Group& group2)
Intercomm Intercomm::Create(const Group& group) const
Intercomm Intracomm::Create_intercomm(int local_leader, const
Comm& peer_comm, int remote_leader, int tag) const
static int Comm::Create_keyval(Comm::Copy_attr_function* comm_copy_attr_fn,
Comm::Delete_attr_function* comm_delete_attr_fn,
void* extra_state)
static int Datatype::Create_keyval(Datatype::Copy_attr_function*
type_copy_attr_fn, Datatype::Delete_attr_function*
type_delete_attr_fn, void* extra_state)
};

A.4.4 Groups, Contexts, Communicators, and Caching C++ Bindings
namespace MPI {
Comm& Comm::Clone() const = 0
Cartcomm& Cartcomm::Clone() const
Graphcomm& Graphcomm::Clone() const
Intercomm& Intercomm::Clone() const
static int Comm::Compare(const Comm& comm1, const Comm& comm2)
static int Group::Compare(const Group& group1, const Group& group2)
Intercomm Intercomm::Create(const Group& group) const
Intercomm Intracomm::Create_intercomm(int local_leader, const
Comm& peer_comm, int remote_leader, int tag) const
static int Comm::Create_keyval(Comm::Copy_attr_function* comm_copy_attr_fn,
Comm::Delete_attr_function* comm_delete_attr_fn,
void* extra_state)
static int Datatype::Create_keyval(Datatype::Copy_attr_function*
type_copy_attr_fn, Datatype::Delete_attr_function*
type_delete_attr_fn, void* extra_state)
A.4. C++ BINDINGS

```cpp
static int Win::Create_keyval(Win::Copy_attr_function* win_copy_attr_fn,
   Win::Delete_attr_function* win_delete_attr_fn,
   void* extra_state)
void Comm::Delete_attr(int comm_keyval)
void Datatype::Delete_attr(int type_keyval)
void Win::Delete_attr(int win_keyval)
static Group Group::Difference(const Group& group1, const Group& group2)
Cartcomm Cartcomm::Dup() const
Graphcomm Graphcomm::Dup() const
Intercomm Intercomm::Dup() const
Intracomm Intracomm::Dup() const
Group Group::Excl(int n, const int ranks[]) const
static void Comm::Free_keyval(int comm_keyval)
static void Datatype::Free_keyval(int& type_keyval)
static void Win::Free_keyval(int& win_keyval)
void Comm::Free()
void Group::Free()
bool Comm::Get_attr(int comm_keyval, void* attribute_val) const
bool Datatype::Get_attr(int type_keyval, void* attribute_val) const
bool Win::Get_attr(int win_keyval, void* attribute_val) const
Group Comm::Get_group() const
void Comm::Get_name(char* comm_name, int& resultlen) const
void Datatype::Get_name(char* type_name, int& resultlen) const
void Win::Get_name(char* win_name, int& resultlen) const
int Comm::Get_rank() const
int Group::Get_rank() const
Group Intercomm::Get_remote_group() const
int Intercomm::Get_remote_size() const
int Comm::Get_size() const
int Group::Get_size() const
Group Group::Incl(int n, const int ranks[]) const
static Group Group::Intersect(const Group& group1, const Group& group2)
```
A.4.5 Process Topologies C++ Bindings

namespace MPI {

  void Compute_dims(int nnodes, int ndims, int dims[])
  Cartcomm Intracomm::Create_cart(int ndims, const int dims[],
                                   const bool periods[], bool reorder) const
  Graphcomm Intracomm::Create_graph(int nnodes[], const int index[],
                                     const int edges[], bool reorder) const
  int Cartcomm::Get_cart_rank(const int coords[]) const
  void Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const
  int Cartcomm::Get_dim() const
  void Graphcomm::Get_dims(int nnodes[], int nedges[]) const
  int Graphcomm::Get_neighbors_count(int rank) const
  void Graphcomm::Get_neighbors(int rank, int maxneighbors, int
                                 neighbors[]) const
  void Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
                          int coords[]) const

  A.4.5 Process Topologies C++ Bindings

  namespace MPI {

  void Compute_dims(int nnodes, int ndims, int dims[])
  Cartcomm Intracomm::Create_cart(int ndims, const int dims[],
                                   const bool periods[], bool reorder) const
  Graphcomm Intracomm::Create_graph(int nnodes[], const int index[],
                                     const int edges[], bool reorder) const
  int Cartcomm::Get_cart_rank(const int coords[]) const
  void Cartcomm::Get_coords(int rank, int maxdims, int coords[]) const
  int Cartcomm::Get_dim() const
  void Graphcomm::Get_dims(int nnodes[], int nedges[]) const
  int Graphcomm::Get_neighbors_count(int rank) const
  void Graphcomm::Get_neighbors(int rank, int maxneighbors, int
                                 neighbors[]) const
  void Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
                          int coords[]) const
### A.4. C++ Bindings

```cpp
void Graphcomm::Get_topo(int maxindex, int maxedges, int index[],
                          int edges[]) const
int Comm::Get_topology() const
int Cartcomm::Map(int ndims, const int dims[], const bool periods[])
                     const
int Graphcomm::Map(int nnodes, const int index[], const int edges[])
                     const
void Cartcomm::Shift(int direction, int disp, int &rank_source,
                      int &rank_dest) const
Cartcomm Cartcomm::Sub(const bool remain_dims[]) const
```

### A.4.6 MPI Environment Management C++ Bindings

```cpp
namespace MPI {
    void Comm::Abort(int errorcode)
    int Add_error_class() int Add_error_code(int errorclass)
    void Add_error_string(int errorcode, const char* string)
    void* Alloc_mem(int size, const Info* info)
    void Comm::Call_errhandler(int errorcode) const
    void File::Call_errhandler(int errorcode) const
    void Win::Call_errhandler(int errorcode) const
    static Errhandler Comm::Create_errhandler(Comm::Errhandler_fn* function)
    static Errhandler File::Create_errhandler(File::Errhandler_fn* function)
    static Errhandler Win::Create_errhandler(Win::Errhandler_fn* function)
    void Finalize() void Free_mem(void *base)
    void Errhandler::Free() Errhandler Comm::Get_errhandler() const
    Errhandler File::Get_errhandler() const
    Errhandler Win::Get_errhandler() const
    int Get_error_class(int errorcode)
    void Get_error_string(int errorcode, char* name, int& resultlen)
}
```
ANNEX A. LANGUAGE BINDINGS SUMMARY

void Get_processor_name(char* name, int& resultlen)
void Get_version(int& version, int& subversion)
void Init(int& argc, char**& argv)
void Init()
bool Is_finalized()
bool Is_initialized()
void Comm::Set_errhandler(const Errhandler& errhandler)

void File::Set_errhandler(const Errhandler& errhandler)
void Win::Set_errhandler(const Errhandler& errhandler)
double Wtick()
double Wtime()

A.4.7 The Info Object C++ Bindings

namespace MPI {
    static Info Info::Create()
    void Info::Delete(const char* key)
    Info Info::Dup() const
    void Info::Free()
    bool Info::Get(const char* key, int valuelen, char* value) const
    int Info::Get_nkeys() const
    void Info::Get_nthkey(int n, char* key) const
    bool Info::Get_valuelen(const char* key, int& valuelen) const
    void Info::Set(const char* key, const char* value)
};

A.4.8 Process Creation and Management C++ Bindings

namespace MPI {
    Intercomm Intracomm::Accept(const char* port_name, const Info& info, int root) const
    void Close_port(const char* port_name)
};
A.4. C++ BINDINGS

```cpp
Intercomm Intracomm::Connect(const char* port_name, const Info& info, int root) const

void Comm::Disconnect()static Intercomm Comm::Get_parent()static Intercomm Comm::Join(const int fd)

void Lookup_name(const char* service_name, const Info& info, char* port_name)
void Open_port(const Info& info, char* port_name)
void Publish_name(const char* service_name, const Info& info, const char* port_name)

Intercomm Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const Info& info, int root) const
Intercomm Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const Info& info, int root, int array_of_errcodes[])

Intercomm Intracomm::Spawn_multiple(int count, const char* array_of_commands[], const char** array_of_argv[], const int array_of_maxprocs[], const Info array_of_info[], int root, int array_of_errcodes[])
Intercomm Intracomm::Spawn_multiple(int count, const char* array_of_commands[], const char** array_of_argv[], const int array_of_maxprocs[], const Info array_of_info[], int root)

void Unpublish_name(const char* service_name, const Info& info, const char* port_name)
```

A.4.9 One-Sided Communications C++ Bindings

```cpp
namespace MPI {

void Win::Accumulate(const void* origin_addr, int origin_count, const Datatype& origin_datatype, int target_rank, Aint target_disp, int target_count, const Datatype& target_datatype, constOp& op) const

void Win::Complete() const
static Win Win::Create(const void* base, Aint size, int disp_unit, const Info& info, const Intracomm& comm)

void Win::Fence(int assert) const
}
```

A.4. C++ BINDINGS

```cpp
Intercomm Intracomm::Connect(const char* port_name, const Info& info, int root) const

void Comm::Disconnect()static Intercomm Comm::Get_parent()static Intercomm Comm::Join(const int fd)

void Lookup_name(const char* service_name, const Info& info, char* port_name)
void Open_port(const Info& info, char* port_name)
void Publish_name(const char* service_name, const Info& info, const char* port_name)

Intercomm Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const Info& info, int root) const
Intercomm Intracomm::Spawn(const char* command, const char* argv[], int maxprocs, const Info& info, int root, int array_of_errcodes[])

Intercomm Intracomm::Spawn_multiple(int count, const char* array_of_commands[], const char** array_of_argv[], const int array_of_maxprocs[], const Info array_of_info[], int root, int array_of_errcodes[])
Intercomm Intracomm::Spawn_multiple(int count, const char* array_of_commands[], const char** array_of_argv[], const int array_of_maxprocs[], const Info array_of_info[], int root)

void Unpublish_name(const char* service_name, const Info& info, const char* port_name)
```
void Win::Free()
Group Win::Get_group() const
void Win::Get(void *origin_addr, int origin_count, const Datatype &
    origin_datatype, int target_rank, Aint target_disp, int
    target_count, const Datatype & target_datatype) const
void Win::Lock(int lock_type, int rank, int assert) const
void Win::Post(const Group & group, int assert) const
void Win::Put(const void* origin_addr, int origin_count, const Datatype &
    origin_datatype, int target_rank, Aint target_disp, int
    target_count, const Datatype & target_datatype) const
void Win::Start(const Group & group, int assert) const
bool Win::Test() const
void Win::Unlock(int rank) const
void Win::Wait() const

A.4.10 External Interfaces C++ Bindings
namespace MPI {
    void Grequest::Complete()
    int Init_thread(int & argc, char** & argv, int required)
    int Init_thread(int required)
    bool Is_thread_main()
    int Query_thread()
    void Status::Set_cancelled(bool flag)
    void Status::Set_elements(const Datatype & datatype, int count)
    static Grequest Grequest::Start(const Grequest::Query_function query_fn,
        const Grequest::Free_function free_fn,
        const Grequest::Cancel_function cancel_fn, void *extra_state)
};

A.4.11 I/O C++ Bindings
namespace MPI {
};
void File::Close()
static void File::Delete(const char* filename, const Info& info)
int File::Get_amode() const
bool File::Get_atomicity() const
Offset File::Get_byte_offset(const Offset disp) const
Group File::Get_group() const
Info File::Get_info() const
Offset File::Get_position() const
Offset File::Get_position_shared() const
Offset File::Get_size() const
int File::Get_type_extent(const Datatype& datatype) const
void File::Get_view(Offset& disp, Datatype& etype, Datatype& filetype, char* datarep) const
Request File::Iread_at(Offset offset, void* buf, int count, const Datatype& datatype)
Request File::Iread_shared(void* buf, int count, const Datatype& datatype)
Request File::Iwrite_at(Offset offset, const void* buf, int count, const Datatype& datatype)
Request File::Iwrite(const void* buf, int count, const Datatype& datatype)
Request File::Iwrite_shared(const void* buf, int count, const Datatype& datatype)
static File File::Open(const Intracomm& comm, const char* filename, int amode, const Info& info)
void File::Preallocate(Offset size)
void File::Read_all_begin(void* buf, int count, const Datatype& datatype)
void File::Read_all_end(void* buf, Status& status)
void File::Read_all(void* buf, int count, const Datatype& datatype, Status& status)
void File::Read_all(void* buf, int count, const Datatype& datatype)
void File::Read_at_all_begin(Offset offset, void* buf, int count, const Datatype& datatype)
void File::Read_at_all_end(void* buf, Status& status)
void File::Read_at_all_end(void* buf)
void File::Read_at_all(Offset offset, void* buf, int count, const Datatype& datatype, Status& status)
void File::Read_at_all(Offset offset, void* buf, int count, const Datatype& datatype)
void File::Read_at(Offset offset, void* buf, int count, const Datatype& datatype, Status& status)
void File::Read_at(Offset offset, void* buf, int count, const Datatype& datatype)
void File::Read_ordered_begin(void* buf, int count, const Datatype& datatype)
void File::Read_ordered_end(void* buf, Status& status)
void File::Read_ordered_end(void* buf)
void File::Read_ordered(void* buf, int count, const Datatype& datatype, Status& status)
void File::Read_ordered(void* buf, int count, const Datatype& datatype)
void File::Read_shared(void* buf, int count, const Datatype& datatype, Status& status)
void File::Read_shared(void* buf, int count, const Datatype& datatype)
void File::Read(void* buf, int count, const Datatype& datatype, Status& status)
void File::Read(void* buf, int count, const Datatype& datatype)
void File::Register_datarep(const char* datarep, Datarep_conversion_function* read_conversion_fn, Datarep_conversion_function* write_conversion_fn, Datarep_extent_function* dtype_file_extent_fn, void* extra_state)
void File::Seek(Offset offset, int whence)
void File::Seek_shared(Offset offset, int whence)
void File::Set_atomicity(bool flag)
void File::Set_info(const Info& info)
void File::Set_size(Offset size)
A.4. C++ BINDINGS

void File::Set_view(Offset disp, const Datatype& etype,
                     const Datatype& filetype, const char* datarep,
                     const Info& info)

void File::Sync()

void File::Write_all_begin(const void* buf, int count,
                           const Datatype& datatype)

void File::Write_all(const void* buf, int count,
                     const Datatype& datatype, Status& status)

void File::Write_all(const void* buf, int count,
                     const Datatype& datatype)

void File::Write_all_end(const void* buf, Status& status)

void File::Write_all_end(const void* buf)

void File::Write_at_all_begin(Offset offset, const void* buf, int count,
                              const Datatype& datatype)

void File::Write_at_all_end(const void* buf, Status& status)

void File::Write_at_all_end(const void* buf)

void File::Write_at_all(Offset offset, const void* buf, int count,
                        const Datatype& datatype, Status& status)

void File::Write_at_all(Offset offset, const void* buf, int count,
                        const Datatype& datatype)

void File::Write_at(Offset offset, const void* buf, int count,
                    const Datatype& datatype, Status& status)

void File::Write_at(Offset offset, const void* buf, int count,
                    const Datatype& datatype)

void File::Write(const void* buf, int count, const Datatype& datatype,
                 Status& status)

void File::Write(const void* buf, int count, const Datatype& datatype)

void File::Write_ordered_begin(const void* buf, int count,
                               const Datatype& datatype)

void File::Write_ordered(const void* buf, int count,
                         const Datatype& datatype, Status& status)

void File::Write_ordered(const void* buf, int count,
                         const Datatype& datatype)

void File::Write_ordered_end(const void* buf, Status& status)

void File::Write_ordered_end(const void* buf)
A.4.12 Language Bindings C++ Bindings
namespace MPI {
    static Datatype Datatype::Create_f90_complex(int p, int r)
    static Datatype Datatype::Create_f90_integer(int r)
    static Datatype Datatype::Create_f90_real(int p, int r)
    Exception::Exception(int error_code)
    int Exception::Get_error_class() const
    int Exception::Get_error_code() const
    const char* Exception::Get_error_string() const
    static Datatype Datatype::Match_size(int typeclass, int size)
};

A.4.13 Profiling Interface C++ Bindings
namespace MPI {
    void Pcontrol(const int level, ...)
};

A.4.14 Deprecated C++ Bindings
namespace MPI {
};

A.4.15 C++ Bindings on all MPI Classes
The C++ language requires all classes to have four special functions: a default constructor, a copy constructor, a destructor, and an assignment operator. The bindings for these functions are listed below; their semantics are discussed in Section 16.1.5. The two constructors are not virtual. The bindings prototype functions are using the type ⟨CLASS⟩ rather than listing each function for every MPI class. The token ⟨CLASS⟩ can be replaced with valid MPI-2 class names, such as Group, Datatype, etc., except when noted. In addition, bindings are provided for comparison and inter-language operability from Sections 16.1.5 and 16.1.9.
### A.4.16 Construction / Destruction

```cpp
namespace MPI {
  ⟨CLASS⟩::⟨CLASS⟩()
  ⟨CLASS⟩::~⟨CLASS⟩()
};
```

### A.4.17 Copy / Assignment

```cpp
namespace MPI {
  ⟨CLASS⟩::⟨CLASS⟩(const ⟨CLASS⟩& data)
  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const ⟨CLASS⟩& data)
};
```

### A.4.18 Comparison

Since `Status` instances are not handles to underlying MPI objects, the `operator==()` and `operator!=()` functions are not defined on the `Status` class.

```cpp
namespace MPI {
  bool ⟨CLASS⟩::operator==(const ⟨CLASS⟩& data) const
  bool ⟨CLASS⟩::operator!=(const ⟨CLASS⟩& data) const
};
```

### A.4.19 Inter-language Operability

Since there are no C++ `MPI::STATUS_IGNORE` and `MPI::STATUSES_IGNORE` objects, the result of promoting the C or Fortran handles (`MPI::STATUS_IGNORE` and `MPI::STATUSES_IGNORE`) to C++ is undefined.

```cpp
namespace MPI {
  ⟨CLASS⟩& ⟨CLASS⟩::operator=(const MPI_⟨CLASS⟩& data)
  ⟨CLASS⟩::operator MPI_⟨CLASS⟩() const
};
```
Annex B

Change-Log

This annex summarizes changes from the previous version of the MPI standard to the version presented by this document. Only changes (i.e., clarifications and new features) are presented that may cause implementation effort in the MPI libraries. Editorial modifications, formatting, typo corrections and minor clarifications are not shown.

B.1 Changes from Version 2.0 to Version 2.1

1. Section 3.2.2 on page 27, Section 16.1.6 on page 453, and Annex A.1 on page 491. In addition, the MPI_LONG_LONG should be added as an optional type; it is a synonym for MPI_LONG_LONG_INT.

2. Section 3.2.2 on page 27, Section 16.1.6 on page 453, and Annex A.1 on page 491. 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.

3. Section 3.2.5 on page 31. 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.

4. Section 4.1 on page 77. General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are nonnegative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

5. Section 4.3 on page 127. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

6. Section 5.9.6 on page 171. If comm is an intercommunicator 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).
B.1. CHANGES FROM VERSION 2.0 TO VERSION 2.1

7. Section 6.3.1 on page 186. 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.

8. Section 6.7 on page 221.

About the attribute caching functions:

Advice to implementors. High-quality implementations should raise an error if a keyval was created by a call to MPI_CONFLICT_SET_ATTR, MPI_CONFLICT_DELETE_ATTR, or MPI_CONFLICT_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.)

9. Section 6.8 on page 235. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger than MPI_MAX_OBJECT-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger than MPI_MAX_OBJECT.

10. Section 7.4 on page 243.

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. Section 7.5.1 on page 244.

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.

12. Section 7.5.3 on page 246.

In MPI_GRAPH_CREATE: The graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes.

13. Section 7.5.3 on page 246.

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 non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric 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.)

14. Section 7.5.4 on page 248.

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.

7. Section 6.3.1 on page 186. 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.

8. Section 6.7 on page 221.

About the attribute caching functions:

Advice to implementors. High-quality implementations should raise an error if a keyval was created by a call to MPI_CONFLICT_SET_ATTR, MPI_CONFLICT_DELETE_ATTR, or MPI_CONFLICT_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.)

9. Section 6.8 on page 235. In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger than MPI_MAX_OBJECT-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger than MPI_MAX_OBJECT.

10. Section 7.4 on page 243.

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. Section 7.5.1 on page 244.

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.

12. Section 7.5.3 on page 246.

In MPI_GRAPH_CREATE: The graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes.

13. Section 7.5.3 on page 246.

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 non-symmetric.

Advice to users. Performance implications of using multiple edges or a non-symmetric 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.)

14. Section 7.5.4 on page 248.

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.
15. Section 7.5.4 on page 248.
   In MPI_CART_RANK: If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank.

16. Section 7.5.4 on page 248.
   In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

17. Section 7.5.5 on page 252.
   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.

18. Section 7.5.6 on page 254.
   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.

19. Section 8.1.2 on page 260.
   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.

20. Section 8.3 on page 264.
   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. Section 8.7 on page 278, 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 10.5.4 on page 318.

22. Section 8.7 on page 278.
   About MPI_ABORT:

   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.)
23. Section 9 on page 287.
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.

24. Section 11.3 on page 325.
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.

25. Section 11.3 on page 325.
After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch. See also item 24 in this list.

26. Section 11.3.4 on page 331.
MPI_REPLACE in MPI_ACCUMULATE, like the other predefined operations, is defined only for the predefined MPI datatypes.

27. Section 13.2.8 on page 382.
About MPI_FILE_SET_VIEW and MPI_FILE_GET_INFO: When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_GET_INFO, there will be no effect on previously set or defaulted hints that the info object does not specify.

28. Section 13.2.8 on page 382.
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.

29. Section 13.3 on page 385.
If a file does not have the mode MPI_MODESEQUENTIAL, then MPI_DISPLACEMENT_CURRENT is invalid as disp in MPI_FILE_SET_VIEW.

30. Section 13.5.2 on page 414.
The bias of 16 byte doubles was defined with 10383. The correct value is 16383.

31. Section 16.1.4 on page 450.
In the example in this section, the buffer should be declared as const void* buf.

32. Section 16.2.5 on page 470.
About MPI_TYPE_CREATE_F90_xx:

Advisors to implementors. An application may often repeat a call to MPI_TYPE_CREATE_F90_xx with the same combination of (xx, 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, the 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_xxxx and using a hash-table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (xxxx, p, r). (End of advice to implementors.)

33. Section A.1.1 on page 491.
   MPI_BOTTOM is defined as void * const MPI::BOTTOM.
Bibliography


Bibliography


## 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/C++ examples.

| Attributes between languages, 487 |
| C++ declarations in mpi.h, 460 |
| C++ deriving from C++ MPI class, 451 |
| C++ handle assignment and comparison operators, 456 |
| C++ handle assignment operator, 452 |
| C++ handle scope destruction, 452 |
| C++ illegal communicator handle initialization, 456 |
| C++ MPI class comparison operator, 451 |
| C++ profiling example, 460, 461 |
| C/C++ handle conversion, 481, 482 |
| Client-server code, 62, 63 |
| Client-server code with blocking probe, 66 |
| Client-server code with blocking probe, wrong, 66 |
| Datatype - 3D array, 111 |
| Datatype - absolute addresses, 117 |
| Datatype - array of structures, 114 |
| Datatype - elaborate example, 124, 125 |
| Datatype - matching type, 101 |
| Datatype - matrix transpose, 112, 113 |
| Datatype - union, 118 |
| Datatypes - matching, 35 |
| Datatypes - not matching, 35 |
| Datatypes - unnamed, 35 |
| Deadlock with MPI_Bcast, 177, 178 |
| Deadlock, if not buffered, 44 |
| Deadlock, wrong message exchange, 44 |
| Fortran 90 copying and sequence problem, 463, 464 |
| Fortran 90 derived types, 465 |
| Fortran 90 heterogeneous communication, 476, 477 |
| Fortran 90 Illegal KIND, 472 |
| Fortran 90 MPI_TYPE_MATCH_SIZE implementation, 476 |
| Fortran 90 register optimization, 467 |
| Intercommunicator, 198, 199 |
| Interlanguage communication, 489 |
| Intertwined matching pairs, 42 |
| Message exchange, 43 |
| MPI::Comm::Probe, 33 |
| MPL_ACCUMULATE, 332 |
| MPL_ADDRESS, 94 |
| MPL_Address, 114, 117, 118, 124 |
| MPL_Aint, 114 |
| MPL_Allgather, 154 |
| MPL_ALLOC_MEM, 263 |
| MPL_ALLOCBUF, 280 |
| MPL_ALLREDUCE, 172 |
| MPL_Barrier, 280, 281, 347 |
| MPL_Bcast, 136, 177, 178 |
| MPL_BSEND, 42 |
| MPL_Buffer_attach, 46, 280 |
| MPL_Buffer_detach, 46 |
| MPL_BYTE, 35 |
| MPL_Cancel, 281 |
| MPL_CART_COORDS, 253 |
| MPL_CART_GET, 256 |
| MPL_CART_RANK, 256 |
| MPL_CART_SHIFT, 253 |
| MPL_CART_SUB, 254 |
| MPL_CHARACTER, 36 |

## 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/C++ examples.

<p>| Attributes between languages, 487 |
| C++ declarations in mpi.h, 460 |
| C++ deriving from C++ MPI class, 451 |
| C++ handle assignment and comparison operators, 456 |
| C++ handle assignment operator, 452 |
| C++ handle scope destruction, 452 |
| C++ illegal communicator handle initialization, 456 |
| C++ MPI class comparison operator, 451 |
| C++ profiling example, 460, 461 |
| C/C++ handle conversion, 481, 482 |
| Client-server code, 62, 63 |
| Client-server code with blocking probe, 66 |
| Client-server code with blocking probe, wrong, 66 |
| Datatype - 3D array, 111 |
| Datatype - absolute addresses, 117 |
| Datatype - array of structures, 114 |
| Datatype - elaborate example, 124, 125 |
| Datatype - matching type, 101 |
| Datatype - matrix transpose, 112, 113 |
| Datatype - union, 118 |
| Datatypes - matching, 35 |
| Datatypes - not matching, 35 |
| Datatypes - unnamed, 35 |
| Deadlock with MPI_Bcast, 177, 178 |
| Deadlock, if not buffered, 44 |
| Deadlock, wrong message exchange, 44 |
| Fortran 90 copying and sequence problem, 463, 464 |
| Fortran 90 derived types, 465 |
| Fortran 90 heterogeneous communication, 476, 477 |
| Fortran 90 Illegal KIND, 472 |
| Fortran 90 MPI_TYPE_MATCH_SIZE implementation, 476 |
| Fortran 90 register optimization, 467 |
| Intercommunicator, 198, 199 |
| Interlanguage communication, 489 |
| Intertwined matching pairs, 42 |
| Message exchange, 43 |
| MPI::Comm::Probe, 33 |
| MPL_ACCUMULATE, 332 |
| MPL_ADDRESS, 94 |
| MPL_Address, 114, 117, 118, 124 |
| MPL_Aint, 114 |
| MPL_Allgather, 154 |
| MPL_ALLOC_MEM, 263 |
| MPL_ALLOCBUF, 280 |
| MPL_ALLREDUCE, 172 |
| MPL_Barrier, 280, 281, 347 |
| MPL_Bcast, 136, 177, 178 |
| MPL_BSEND, 42 |
| MPL_Buffer_attach, 46, 280 |
| MPL_Buffer_detach, 46 |
| MPL_BYTE, 35 |
| MPL_Cancel, 281 |
| MPL_CART_COORDS, 253 |
| MPL_CART_GET, 256 |
| MPL_CART_RANK, 256 |
| MPL_CART_SHIFT, 253 |
| MPL_CART_SUB, 254 |
| MPL_CHARACTER, 36 |</p>
<table>
<thead>
<tr>
<th>Examples Index</th>
<th>573</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_start, 339, 347</td>
<td>1</td>
</tr>
<tr>
<td>MPI_Win_unlock, 344</td>
<td>2</td>
</tr>
<tr>
<td>MPI_Win_wait, 347</td>
<td>3</td>
</tr>
<tr>
<td>mpiexec, 286</td>
<td>4</td>
</tr>
<tr>
<td>Non-deterministic program with MPI_Bcast, 178</td>
<td>5</td>
</tr>
<tr>
<td>Non-overtaking messages, 42</td>
<td>6</td>
</tr>
<tr>
<td>Nonblocking operations, 54, 55</td>
<td>7</td>
</tr>
<tr>
<td>Nonblocking operations - message ordering, 56</td>
<td>8</td>
</tr>
<tr>
<td>Nonblocking operations - progress, 56</td>
<td>9</td>
</tr>
<tr>
<td>Threads and MPI, 366</td>
<td>10</td>
</tr>
<tr>
<td>Typemap, 79–81, 83, 86, 93</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples Index</th>
<th>573</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_start, 339, 347</td>
<td>1</td>
</tr>
<tr>
<td>MPI_Win_unlock, 344</td>
<td>2</td>
</tr>
<tr>
<td>MPI_Win_wait, 347</td>
<td>3</td>
</tr>
<tr>
<td>mpiexec, 286</td>
<td>4</td>
</tr>
<tr>
<td>Non-deterministic program with MPI_Bcast, 178</td>
<td>5</td>
</tr>
<tr>
<td>Non-overtaking messages, 42</td>
<td>6</td>
</tr>
<tr>
<td>Nonblocking operations, 54, 55</td>
<td>7</td>
</tr>
<tr>
<td>Nonblocking operations - message ordering, 56</td>
<td>8</td>
</tr>
<tr>
<td>Nonblocking operations - progress, 56</td>
<td>9</td>
</tr>
<tr>
<td>Threads and MPI, 366</td>
<td>10</td>
</tr>
<tr>
<td>Typemap, 79–81, 83, 86, 93</td>
<td>11</td>
</tr>
</tbody>
</table>
### MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

<table>
<thead>
<tr>
<th>Constant/Handle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI:*_NULL</td>
<td>451</td>
</tr>
<tr>
<td>MPI:BOOL</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:BYTE</td>
<td>453-455</td>
</tr>
<tr>
<td>MPI:CHAR</td>
<td>454</td>
</tr>
<tr>
<td>MPI:COMM_NULL</td>
<td>453, 456</td>
</tr>
<tr>
<td>MPI:COMPLEX</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:DOUBLE</td>
<td>454</td>
</tr>
<tr>
<td>MPI:DOUBLE_COMPLEX</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:DOUBLE_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:DOUBLE_PRECISION</td>
<td>454</td>
</tr>
<tr>
<td>MPI:ERRORS</td>
<td>19</td>
</tr>
<tr>
<td>MPI:ERRORS_THROW_EXCEPTIONS</td>
<td>19, 23, 265</td>
</tr>
<tr>
<td>MPI:F_COMPLEX</td>
<td>454</td>
</tr>
<tr>
<td>MPI:F_DOUBLE_COMPLEX</td>
<td>455</td>
</tr>
<tr>
<td>MPI:FLOAT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:FLOAT_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:INTEGER1</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER2</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER4</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER8</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:MAXLOC</td>
<td>455</td>
</tr>
<tr>
<td>MPI:MINLOC</td>
<td>455</td>
</tr>
<tr>
<td>MPI:PACKED</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:REAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:REAL16</td>
<td>454</td>
</tr>
<tr>
<td>MPI:REAL4</td>
<td>455</td>
</tr>
<tr>
<td>MPI:REAL8</td>
<td>455</td>
</tr>
<tr>
<td>MPI:REALLOC</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SHORT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SIGNED_CHAR</td>
<td>454</td>
</tr>
<tr>
<td>MPI:UNSIGNED_CHAR</td>
<td>453</td>
</tr>
<tr>
<td>MPI:TWODoublePRECISION</td>
<td>455</td>
</tr>
<tr>
<td>MPI:TWODINT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:TWOREAL</td>
<td>455</td>
</tr>
<tr>
<td>MPI: UNSIGNED_LONG</td>
<td>454</td>
</tr>
<tr>
<td>MPI: UNSIGNED_LONG_LONG</td>
<td>454</td>
</tr>
<tr>
<td>MPI: UNSIGNED_SHORT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:UNSIGNED_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:WCHAR</td>
<td>454</td>
</tr>
<tr>
<td>MPI:WDOUBLEPRECISION</td>
<td>165</td>
</tr>
<tr>
<td>MPI:WINT</td>
<td>165</td>
</tr>
<tr>
<td>MPI:WINTEGER</td>
<td>165</td>
</tr>
<tr>
<td>MPI:都REAL</td>
<td>165</td>
</tr>
<tr>
<td>MPI:ADDRESS_KIND</td>
<td>18, 106, 486, 487</td>
</tr>
<tr>
<td>MPI:ANY_SOURCE</td>
<td>30, 42, 64-66, 71, 240, 261</td>
</tr>
<tr>
<td>MPI:ANY_TAG</td>
<td>14, 30, 32, 64, 65, 67, 71, 75</td>
</tr>
<tr>
<td>MPI:APPNUM</td>
<td>317, 318</td>
</tr>
<tr>
<td>MPI:ARGV_NULL</td>
<td>298, 299, 462</td>
</tr>
<tr>
<td>MPI:ARGVS_NULL</td>
<td>302, 462</td>
</tr>
<tr>
<td>MPI:BAND</td>
<td>161, 162</td>
</tr>
<tr>
<td>MPI:BOR</td>
<td>162</td>
</tr>
</tbody>
</table>

### MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

<table>
<thead>
<tr>
<th>Constant/Handle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI:*_NULL</td>
<td>451</td>
</tr>
<tr>
<td>MPI:BOOL</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:BYTE</td>
<td>453-455</td>
</tr>
<tr>
<td>MPI:CHAR</td>
<td>454</td>
</tr>
<tr>
<td>MPI:COMM_NULL</td>
<td>453, 456</td>
</tr>
<tr>
<td>MPI:COMPLEX</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:DOUBLE</td>
<td>454</td>
</tr>
<tr>
<td>MPI:DOUBLE_COMPLEX</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:DOUBLE_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:DOUBLE_PRECISION</td>
<td>454</td>
</tr>
<tr>
<td>MPI:ERRORS</td>
<td>19</td>
</tr>
<tr>
<td>MPI:ERRORS_THROW_EXCEPTIONS</td>
<td>19, 23, 265</td>
</tr>
<tr>
<td>MPI:F_COMPLEX</td>
<td>454</td>
</tr>
<tr>
<td>MPI:F_DOUBLE_COMPLEX</td>
<td>455</td>
</tr>
<tr>
<td>MPI:FLOAT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:FLOAT_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:INTEGER1</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER2</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER4</td>
<td>455</td>
</tr>
<tr>
<td>MPI:INTEGER8</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:MAXLOC</td>
<td>455</td>
</tr>
<tr>
<td>MPI:MINLOC</td>
<td>455</td>
</tr>
<tr>
<td>MPI:PACKED</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:REAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:REAL16</td>
<td>454</td>
</tr>
<tr>
<td>MPI:REAL4</td>
<td>455</td>
</tr>
<tr>
<td>MPI:REAL8</td>
<td>455</td>
</tr>
<tr>
<td>MPI:REALLOC</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SHORT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SIGNED_CHAR</td>
<td>454</td>
</tr>
<tr>
<td>MPI:UNSIGNED_CHAR</td>
<td>453</td>
</tr>
<tr>
<td>MPI:TWODoublePRECISION</td>
<td>455</td>
</tr>
<tr>
<td>MPI:TWODINT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:TWOREAL</td>
<td>455</td>
</tr>
<tr>
<td>MPI: NDEBUG_LONG</td>
<td>454</td>
</tr>
<tr>
<td>MPI: NDEBUG_LONG_LONG</td>
<td>454</td>
</tr>
<tr>
<td>MPI: NDEBUG_SHORT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:WCHAR</td>
<td>454</td>
</tr>
<tr>
<td>MPI:WDOUBLEPRECISION</td>
<td>165</td>
</tr>
<tr>
<td>MPI:WINT</td>
<td>165</td>
</tr>
<tr>
<td>MPI:WINTEGER</td>
<td>165</td>
</tr>
<tr>
<td>MPI:都REAL</td>
<td>165</td>
</tr>
<tr>
<td>MPI:ADDRESS_KIND</td>
<td>18, 106, 486, 487</td>
</tr>
<tr>
<td>MPI:ANY_SOURCE</td>
<td>30, 42, 64-66, 71, 240, 261</td>
</tr>
<tr>
<td>MPI:ANY_TAG</td>
<td>14, 30, 32, 64, 65, 67, 71, 75</td>
</tr>
<tr>
<td>MPI:APPNUM</td>
<td>317, 318</td>
</tr>
<tr>
<td>MPI:ARGV_NULL</td>
<td>298, 299, 462</td>
</tr>
<tr>
<td>MPI:ARGVS_NULL</td>
<td>302, 462</td>
</tr>
<tr>
<td>MPI:BAND</td>
<td>161, 162</td>
</tr>
<tr>
<td>MPI:BOR</td>
<td>162</td>
</tr>
</tbody>
</table>
MPI Declarations Index

This index refers to declarations needed in C/C++, 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).

MPI::Aint, 15, 15, 19, 79, 79, 81, 84, 86, 94, 97, 98, 107, 127, 128, 322, 326, 328, 331, 413, 417, 441–444, 485, 486
MPI::Cartcomm, 244, 450, 456
MPI::Comm, 26, 188, 193–196, 198, 201, 210, 211, 213, 223, 226, 227, 450, 456, 456
MPI::Datatype, 79, 450
MPI::Errhandler, 266, 267–270, 446, 447, 450, 480
MPI::Exception, 450, 457
MPI::Graphcomm, 246, 450, 456
MPI::Grequest, 358, 358, 450
MPI::Group, 186, 186, 187–192, 196, 211, 324, 329, 340, 380, 450, 480
MPI::Intercomm, 450, 456
MPI::Intracomm, 450, 456
MPI::Offset, 15, 15, 19, 379, 380, 385, 387, 391–393, 398, 399, 403, 404, 406, 407, 417, 424
MPI::Op, 160, 168, 170, 172–175, 231, 450, 480
MPI::Prequest, 69, 450
MPI::Request, 49, 51, 53, 54, 57–61, 64, 67, 69–72, 358, 361, 393, 397, 398, 401, 450, 480
MPI::Status, 29, 32, 53, 57–61, 64, 65, 68, 73, 74, 102, 358, 364, 391, 392, 394–396, 400, 402, 403, 406–410, 450, 451, 482, 483
MPI::Win, 228–230, 238, 268, 276, 322, 332, 346, 328, 331, 413, 417, 441–444, 485, 486
MPI::Cartcomm, 244, 440, 446
MPI::Comm, 26, 188, 193–196, 198, 201, 210, 211, 213, 223, 226, 227, 450, 456, 456
MPI::Datatype, 79, 450
MPI::Errhandler, 266, 267–270, 446, 447, 450, 480
MPI::Exception, 450, 457
MPI::Graphcomm, 246, 450, 456
MPI::Grequest, 358, 358, 450
MPI::Group, 186, 186, 187–192, 196, 211, 324, 339, 340, 380, 450, 480
MPI::Intercomm, 450, 456
MPI::Intracomm, 450, 456
MPI::Offset, 15, 15, 19, 379, 380, 385, 387, 391–393, 398, 399, 403, 404, 406, 407, 417, 424
MPI::Op, 160, 168, 170, 172–175, 231, 450, 480
MPI::Prequest, 69, 450
MPI::Request, 49, 51, 53, 54, 57–61, 64, 67, 69–72, 358, 361, 393, 397, 398, 401, 450, 480
MPI::Status, 29, 32, 53, 57–61, 64, 65, 68, 73, 74, 102, 358, 364, 391, 392, 394–396, 400, 402, 403, 406–410, 482, 483

MPI Declarations Index

This index refers to declarations needed in C/C++, 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).

MPI::Aint, 15, 15, 19, 79, 79, 81, 84, 86, 94, 97, 98, 107, 127, 128, 322, 326, 328, 331, 413, 417, 441–444, 485, 486
MPI::Cartcomm, 244, 440, 446
MPI::Comm, 26, 188, 193–196, 198, 201, 210, 211, 213, 223, 226, 227, 450, 456, 456
MPI::Datatype, 79, 450
MPI::Errhandler, 266, 267–270, 446, 447, 450, 480
MPI::Exception, 450, 457
MPI::Graphcomm, 246, 450, 456
MPI::Grequest, 358, 358, 450
MPI::Group, 186, 186, 187–192, 196, 211, 324, 339, 340, 380, 450, 480
MPI::Intercomm, 450, 456
MPI::Intracomm, 450, 456
MPI::Offset, 15, 15, 19, 379, 380, 385, 387, 391–393, 398, 399, 403, 404, 406, 407, 417, 424
MPI::Op, 160, 168, 170, 172–175, 231, 450, 480
MPI::Prequest, 69, 450
MPI::Request, 49, 51, 53, 54, 57–61, 64, 67, 69–72, 358, 361, 393, 397, 398, 401, 450, 480
MPI::Status, 29, 32, 53, 57–61, 64, 65, 68, 73, 74, 102, 358, 364, 391, 392, 394–396, 400, 402, 403, 406–410, 482, 483

578 578
MPI Declarations Index

MPI_Win, 228–230, 238, 268, 276, 322, 323, 324, 326, 328, 331, 338–343, 480

MPI Declarations Index

MPI_Win, 228–230, 238, 268, 276, 322, 323, 324, 326, 328, 331, 338–343, 480
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. C++ names for these typedefs and Fortran example prototypes are given near the text of the C name.

MPI_Comm_copy_attr_function, 17, 223
MPI_Comm_delete_attr_function, 17, 223
MPI_Comm_errhandler_fn, 17, 266
MPI_Copy_function, 17, 444
MPI_Datarep_conversion_function, 417
MPI_Datarep_extent_function, 417
MPI_Delete_function, 17, 445
MPI_File_errhandler_fn, 269
MPI_Grequest_cancel_function, 360
MPI_Grequest_free_function, 359
MPI_Grequest_query_function, 358
MPI_Handler_function, 17, 447
MPI_Type_copy_attr_function, 231
MPI_Type_delete_attr_function, 231
MPI_User_function, 168
MPI_Win_copy_attr_function, 228
MPI_Win_delete_attr_function, 228
MPI_Win_errhandler_fn, 268

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. C++ names for these typedefs and Fortran example prototypes are given near the text of the C name.

MPI_Comm_copy_attr_function, 17, 223
MPI_Comm_delete_attr_function, 17, 223
MPI_Comm_errhandler_fn, 17, 266
MPI_Copy_function, 17, 444
MPI_Datarep_conversion_function, 417
MPI_Datarep_extent_function, 417
MPI_Delete_function, 17, 445
MPI_File_errhandler_fn, 269
MPI_Grequest_cancel_function, 360
MPI_Grequest_free_function, 359
MPI_Grequest_query_function, 358
MPI_Handler_function, 17, 447
MPI_Type_copy_attr_function, 231
MPI_Type_delete_attr_function, 231
MPI_User_function, 168
MPI_Win_copy_attr_function, 228
MPI_Win_delete_attr_function, 228
MPI_Win_errhandler_fn, 268
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_WIN_CREATE_KEYVAL</td>
<td>222, 228, 233, 486, 563</td>
</tr>
<tr>
<td>MPI_WIN_DELETE_ATTR</td>
<td>222, 230, 233</td>
</tr>
<tr>
<td>MPI_WIN_DUP_FN</td>
<td>228, 228</td>
</tr>
<tr>
<td>MPI_WIN_F2C</td>
<td>480</td>
</tr>
<tr>
<td>MPI_WIN_FENCE</td>
<td>324, 334, 338, 338, 344, 349-352</td>
</tr>
<tr>
<td>MPI_WIN_FREE</td>
<td>229, 319, 323, 324</td>
</tr>
<tr>
<td>MPI_WIN_GET_ATTR</td>
<td>222, 224, 230, 233, 324, 488</td>
</tr>
<tr>
<td>MPI_WIN_GET_ERRHANDLER</td>
<td>265, 268, 564</td>
</tr>
<tr>
<td>MPI_WIN_GET_GROUP</td>
<td>324, 325</td>
</tr>
<tr>
<td>MPI_WIN_GET_NAME</td>
<td>238</td>
</tr>
<tr>
<td>MPI_WIN_KEYVAL_CREATE</td>
<td>501</td>
</tr>
<tr>
<td>MPI_WIN_LOCK</td>
<td>262, 323, 334, 343, 344, 350</td>
</tr>
<tr>
<td>MPI_WIN_NULL_COPY_FN</td>
<td>228, 228</td>
</tr>
<tr>
<td>MPI_WIN_NULL_DELETE_FN</td>
<td>228</td>
</tr>
<tr>
<td>MPI_WIN_POST</td>
<td>324, 334, 339, 340, 340, 341-345, 352</td>
</tr>
<tr>
<td>MPI_WIN_POST_MPI_WIN_FENCE</td>
<td>350</td>
</tr>
<tr>
<td>MPI_WIN_SET_ATTR</td>
<td>222, 229, 233</td>
</tr>
<tr>
<td>MPI_WIN_SET_ERRHANDLER</td>
<td>265, 268</td>
</tr>
<tr>
<td>MPI_WIN_SET_NAME</td>
<td>238</td>
</tr>
<tr>
<td>MPI_WIN_START</td>
<td>334, 339, 340, 340, 342, 344, 345, 348</td>
</tr>
<tr>
<td>MPI_WIN_TEST</td>
<td>341, 341</td>
</tr>
<tr>
<td>MPI_WIN_UNLOCK</td>
<td>262, 323, 334, 344, 349, 350, 352</td>
</tr>
<tr>
<td>MPI_WIN_WAIT</td>
<td>324, 334, 340, 340, 341-343, 350, 352</td>
</tr>
<tr>
<td>MPI_WTICK</td>
<td>21, 278, 278</td>
</tr>
<tr>
<td>MPI_WTIME</td>
<td>21, 261, 277, 277, 278</td>
</tr>
<tr>
<td>mpiexec</td>
<td>279, 283, 285, 369</td>
</tr>
<tr>
<td>mpirun</td>
<td>284, 285</td>
</tr>
</tbody>
</table>

**Notes:**
- MPI: MPI Function Index
- MPI_WIN: Use with Windows
- PMPI: Use with Parallel MPI
- MPPL: Use with Parallel MPI (Linux)

**Page Numbers:**
- 222, 228, 233, 486, 563
- 222, 230, 233
- 228, 228
- 324, 334, 338, 338
- 344, 349-352
- 229, 319, 323, 324
- 222, 224, 230, 233, 324, 488
- 265, 268, 564
- 324, 325
- 238
- 501
- 262, 323, 334, 343, 344, 350
- 228, 228
- 228
- 324, 334, 339, 340, 340, 341-345, 352
- 222, 229, 233
- 265, 268
- 238
- 334, 339, 340, 342, 344, 345, 348
- 341, 341
- 262, 323, 334, 344, 349, 350, 352
- 324, 334, 340, 340, 341-343, 350, 352
- 21, 278, 278
- 21, 261, 277, 277, 278
- 279, 283, 285, 369
- 284, 285
- 435
- 21
- 21

**Additional Functions:**
- MPI_WIN_POST: Use with Windows
- MPI_WTICK: Use with Windows
- MPI_WTIME: Use with Windows