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.
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.

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.
5 Collective Communication

6 Groups, Contexts, Communicators, and Caching

4.3 Canonical MPI_PACK and MPI_UNPACK

5 Collective Communication

6 Groups, Contexts, Communicators, and Caching

4.3 Canonical MPI_PACK and MPI_UNPACK
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.2.5 Preallocating Space for a File</td>
<td>379</td>
</tr>
<tr>
<td>13.2.6 Querying the Size of a File</td>
<td>380</td>
</tr>
<tr>
<td>13.2.7 Querying File Parameters</td>
<td>380</td>
</tr>
<tr>
<td>13.2.8 File Info</td>
<td>382</td>
</tr>
<tr>
<td>Reserved File Hints</td>
<td>383</td>
</tr>
<tr>
<td>13.3 File Views</td>
<td>385</td>
</tr>
<tr>
<td>13.4 Data Access</td>
<td>387</td>
</tr>
<tr>
<td>13.4.1 Data Access Routines</td>
<td>387</td>
</tr>
<tr>
<td>Positioning</td>
<td>388</td>
</tr>
<tr>
<td>Synchronism</td>
<td>389</td>
</tr>
<tr>
<td>Coordination</td>
<td>389</td>
</tr>
<tr>
<td>Data Access Conventions</td>
<td>389</td>
</tr>
<tr>
<td>13.4.2 Data Access with Explicit Offsets</td>
<td>390</td>
</tr>
<tr>
<td>13.4.3 Data Access with Individual File Pointers</td>
<td>394</td>
</tr>
<tr>
<td>13.4.4 Data Access with Shared File Pointers</td>
<td>399</td>
</tr>
<tr>
<td>Noncollective Operations</td>
<td>400</td>
</tr>
<tr>
<td>Collective Operations</td>
<td>402</td>
</tr>
<tr>
<td>Seek</td>
<td>403</td>
</tr>
<tr>
<td>13.4.5 Split Collective Data Access Routines</td>
<td>404</td>
</tr>
<tr>
<td>File Interoperability</td>
<td>410</td>
</tr>
<tr>
<td>13.5.1 Datatypes for File Interoperability</td>
<td>412</td>
</tr>
<tr>
<td>13.5.2 External Data Representation: &quot;external32&quot;</td>
<td>414</td>
</tr>
<tr>
<td>13.5.3 User-Defined Data Representations</td>
<td>415</td>
</tr>
<tr>
<td>Extent Callback</td>
<td>417</td>
</tr>
<tr>
<td>Datarep Conversion Functions</td>
<td>417</td>
</tr>
<tr>
<td>13.5.4 Matching Data Representations</td>
<td>419</td>
</tr>
<tr>
<td>Consistency and Semantics</td>
<td>420</td>
</tr>
<tr>
<td>13.6.1 File Consistency</td>
<td>420</td>
</tr>
<tr>
<td>13.6.2 Random Access vs. Sequential Files</td>
<td>423</td>
</tr>
<tr>
<td>13.6.3 Progress</td>
<td>423</td>
</tr>
<tr>
<td>13.6.4 Collective File Operations</td>
<td>423</td>
</tr>
<tr>
<td>13.6.5 Type Matching</td>
<td>424</td>
</tr>
<tr>
<td>13.6.6 Miscellaneous Clarifications</td>
<td>424</td>
</tr>
<tr>
<td>13.6.7 MPI_Offset Type</td>
<td>424</td>
</tr>
<tr>
<td>13.6.8 Logical vs. Physical File Layout</td>
<td>424</td>
</tr>
<tr>
<td>13.6.9 File Size</td>
<td>425</td>
</tr>
<tr>
<td>13.6.10 Examples</td>
<td>425</td>
</tr>
<tr>
<td>Asynchronous I/O</td>
<td>428</td>
</tr>
<tr>
<td>13.7 I/O Error Handling</td>
<td>429</td>
</tr>
<tr>
<td>13.8 I/O Error Classes</td>
<td>430</td>
</tr>
<tr>
<td>13.9 Examples</td>
<td>430</td>
</tr>
<tr>
<td>13.9.1 Double Buffering with Split Collective I/O</td>
<td>430</td>
</tr>
<tr>
<td>13.9.2 Subarray Filetype Constructor</td>
<td>433</td>
</tr>
</tbody>
</table>
14 Profiling Interface
14.1 Requirements ........................................ 435
14.2 Discussion ............................................ 435
14.3 Logic of the Design ................................... 436
14.3.1 Miscellaneous Control of Profiling ............ 436
14.4 Examples .............................................. 437
14.4.1 Profiler Implementation .......................... 437
14.4.2 MPI Library Implementation ..................... 438
Systems with Weak Symbols .............................. 438
Systems Without Weak Symbols ....................... 438
14.4.3 Complications ..................................... 439
Multiple Counting ....................................... 439
Linker Oddities ......................................... 439
14.5 Multiple Levels of Interception .................... 440

15 Deprecated Functions
15.1 Deprecated since MPI-2.0 .............................. 441

16 Language Bindings
16.1 C++ .................................................. 449
16.1.1 Overview ......................................... 449
16.1.2 Design ............................................ 449
16.1.3 C++ Classes for MPI ............................ 450
16.1.4 Class Member Functions for MPI ............... 450
16.1.5 Semantics ........................................ 451
16.1.6 C++ Datatypes ................................... 453
16.1.7 Communicators ................................... 455
16.1.8 Exceptions ....................................... 457
16.1.9 Mixed-Language Operability ..................... 458
16.1.10 Profiling ........................................ 458
16.2 Fortran Support ....................................... 461
16.2.1 Overview ......................................... 461
16.2.2 Problems With Fortran Bindings for MPI .... 462
Problems Due to Strong Typing ......................... 463
Problems Due to Data Copying and Sequence Association 463
Special Constants ...................................... 465
Fortran 90 Derived Types ................................ 465
A Problem with Register Optimization ................ 466
16.2.3 Basic Fortran Support ............................ 468
16.2.4 Extended Fortran Support ....................... 469
The mpi Module .......................................... 469
No Type Mismatch Problems for Subroutines with Choice Arguments 470
16.2.5 Additional Support for Fortran Numeric Intrinsic Types .................................. 470
Parameterized Datatypes with Specified Precision and Exponent Range .................................. 471
Support for Size-specific MPI Datatypes ................ 474
Communication With Size-specific Types ................ 478
16.3 Language Interoperability ........................... 478
16.3.1 Introduction ...................................... 478
14 Profiling Interface
14.1 Requirements ........................................ 435
14.2 Discussion ............................................ 435
14.3 Logic of the Design ................................... 436
14.3.1 Miscellaneous Control of Profiling ............ 436
14.4 Examples .............................................. 437
14.4.1 Profiler Implementation .......................... 437
14.4.2 MPI Library Implementation ..................... 438
Systems with Weak Symbols .............................. 438
Systems Without Weak Symbols ....................... 438
14.4.3 Complications ..................................... 439
Multiple Counting ....................................... 439
Linker Oddities ......................................... 439
14.5 Multiple Levels of Interception .................... 440

15 Deprecated Functions
15.1 Deprecated since MPI-2.0 .............................. 441

16 Language Bindings
16.1 C++ .................................................. 449
16.1.1 Overview ......................................... 449
16.1.2 Design ............................................ 449
16.1.3 C++ Classes for MPI ............................ 450
16.1.4 Class Member Functions for MPI ............... 450
16.1.5 Semantics ........................................ 451
16.1.6 C++ Datatypes ................................... 453
16.1.7 Communicators ................................... 455
16.1.8 Exceptions ....................................... 457
16.1.9 Mixed-Language Operability ..................... 458
16.1.10 Profiling ........................................ 458
16.2 Fortran Support ....................................... 461
16.2.1 Overview ......................................... 461
16.2.2 Problems With Fortran Bindings for MPI .... 462
Problems Due to Strong Typing ......................... 463
Problems Due to Data Copying and Sequence Association 463
Special Constants ...................................... 465
Fortran 90 Derived Types ................................ 465
A Problem with Register Optimization ................ 466
16.2.3 Basic Fortran Support ............................ 468
16.2.4 Extended Fortran Support ....................... 469
The mpi Module .......................................... 469
No Type Mismatch Problems for Subroutines with Choice Arguments 470
16.2.5 Additional Support for Fortran Numeric Intrinsic Types .................................. 470
Parameterized Datatypes with Specified Precision and Exponent Range .................................. 471
Support for Size-specific MPI Datatypes ................ 474
Communication With Size-specific Types ................ 478
16.3 Language Interoperability ........................... 478
16.3.1 Introduction ...................................... 478
List of Figures

5.1 Collective communications, an overview .......................... 130
5.2 Intercommunicator allgather ........................................ 134
5.3 Intercommunicator reduce-scatter .................................. 135
5.4 Gather example .......................................................... 141
5.5 Gather example with strides .......................................... 142
5.6 Gather example, 2-dimensional ...................................... 142
5.7 Gather example, 2-dimensional, subarrays with different sizes .... 143
5.8 Gather example, 2-dimensional, subarrays with different sizes and strides 145
5.9 Scatter example .......................................................... 150
5.10 Scatter example with strides ......................................... 151
5.11 Scatter example with different strides and counts ............... 152
5.12 Race conditions with point-to-point and collective communications 179
6.1 Intercommunicator create using MPI_COMM_CREATE .................. 197
6.2 Intercommunicator construction with MPI_COMM_SPLIT ............. 200
6.3 Three-group pipeline .................................................... 214
6.4 Three-group ring ......................................................... 215
7.1 Set-up of process structure for two-dimensional parallel Poisson solver 257
11.1 Active target communication ......................................... 335
11.2 Active target communication, with weak synchronization ....... 336
11.3 Passive target communication ........................................ 337
11.4 Active target communication with several processes .............. 341
11.5 Schematic description of window ..................................... 350
11.6 Symmetric communication ............................................ 353
11.7 Deadlock situation ....................................................... 353
11.8 No deadlock ............................................................... 354
13.1 Etypes and filetypes ..................................................... 374
13.2 Partitioning a file among parallel processes ....................... 374
13.3 Displacements ........................................................... 386
13.4 Example array file layout ............................................. 433
13.5 Example local array filetype for process 1 .......................... 433
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

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
Acknowledgments

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

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

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

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communications
- Al Geist, Marc Snir, Steve Otto, Collective Communications
- Steve Otto, Editor
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset

The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

Ed Anderson  Robert Babb  Joe Baron  Eric Barszcz
Scott Berryman Bob Bjornson Nathan Doss Anne Elster
Jim Feeney  Vince Fernando Sam Fineberg Jon Flower
Daniel Frye Ian Glendinning Adam Greenberg Robert Harrison
Leslie Hart Tom Haupt Don Heller Tom Henderson
Alex Ho  C.T. Howard Ho Gary Howell John Kapenga
James Kohl  Susan Krauss Bob Leary Arthur MacCabe
Peter Madams Alan Mainwaring Oliver McBryan Phil McKinley
Charles Mosher Dan Nessett Peter Pacheco Howard Palmer
Paul Pierce Sanjay Ranka Peter Rigbee Arch Robison
Erich Schikuta Ambuj Singh Alan Sussman Robert Tomlinson
Robert G. Voigt Dennis Weeks Stephen Wheat Steve Zenith

Acknowledgments

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

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

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

- Jack Dongarra, David Walker, Conveners and Meeting Chairs
- Ewing Lusk, Bob Knighten, Minutes
- Marc Snir, William Gropp, Ewing Lusk, Point-to-Point Communications
- Al Geist, Marc Snir, Steve Otto, Collective Communications
- Steve Otto, Editor
- Rolf Hempel, Process Topologies
- Ewing Lusk, Language Binding
- William Gropp, Environmental Management
- James Cownie, Profiling
- Tony Skjellum, Lyndon Clarke, Marc Snir, Richard Littlefield, Mark Sears, Groups, Contexts, and Communicators
- Steven Huss-Lederman, Initial Implementation Subset

The following list includes some of the active participants in the MPI-1.0 and MPI-1.1 process not mentioned above.

Ed Anderson  Robert Babb  Joe Baron  Eric Barszcz
Scott Berryman Bob Bjornson Nathan Doss Anne Elster
Jim Feeney  Vince Fernando Sam Fineberg Jon Flower
Daniel Frye Ian Glendinning Adam Greenberg Robert Harrison
Leslie Hart Tom Haupt Don Heller Tom Henderson
Alex Ho  C.T. Howard Ho Gary Howell John Kapenga
James Kohl  Susan Krauss Bob Leary Arthur MacCabe
Peter Madams Alan Mainwaring Oliver McBryan Phil McKinley
Charles Mosher Dan Nessett Peter Pacheco Howard Palmer
Paul Pierce Sanjay Ranka Peter Rigbee Arch Robison
Erich Schikuta Ambuj Singh Alan Sussman Robert Tomlinson
Robert G. Voigt Dennis Weeks Stephen Wheat Steve Zenith
MPI-1.2 and MPI-2.0:

Those who served as primary coordinators in MPI-1.2 and MPI-2.0 are:

- Ewing Lusk, Convener and Meeting Chair
- Steve Huss-Lederman, Editor
- Ewing Lusk, Miscellany
- Bill Saphir, Process Creation and Management
- Marc Snir, One-Sided Communications
- Bill Gropp and Anthony Skjellum, Extended Collective Operations
- Steve Huss-Lederman, External Interfaces
- Bill Nitzberg, I/O
- Andrew Lumsdaine, Bill Saphir, and Jeff Squyres, Language Bindings
- Anthony Skjellum and Arkady Kanevsky, Real-Time

The following list includes some of the active participants who attended MPI-2 Forum meetings and are not mentioned above.

Greg Astfalk  Robert Babb  Ed Benson  Rajesh Bordawekar
Pete Bradley  Peter Brennan  Ron Brightwell  Maciej Brodowicz
Eric Brunner  Greg Burns  Margaret Cahir  Puny Chen
Ying Chen  Albert Cheng  Yong Cho  Joel Clark
Lyndon Clarke  Laurie Costello  Dennis Cottel  Jim Cowanie
Zheqian Cui  Suresh Damodaran-Kamal  Raja Daoud
Judith Devaney  David DiNucci  Doug Doeffler  Jack Dongarra
Terry Donije  Nathan Doss  Anne Elster  Mark Fallon
Karli Feind  Sam Fineberg  Craig Fischberg  Stephen Fleischman
Ian Foster  Hubertus Franke  Richard Frost  Al Geist
Robert George  David Greenberg  John Hagedorn  Kei Harada
Leslie Hart  Shane Hebert  Rolf Hempel  Tom Henderson
Alex Ho  Hans-Christian Hoppe  Joeon Jann  Terry Jones
Karl Kesselman  Koichi Konishi  Susan Kraus  Steve Kubica
Steve Landherr  Mario Lauria  Mark Law  Juan Leon
Lloyd Lewins  Ziyang Lu  Bob Madahar  Peter Madams
The MPI Forum also acknowledges and appreciates the valuable input from people via e-mail and in person.

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

- Argonne National Laboratory
- Bolt, Beranek, and Newman
- California Institute of Technology
- Center for Computing Sciences
- Convex Computer Corporation
- Cray Research
- Digital Equipment Corporation
- Dolphin Interconnect Solutions, Inc.
- Edinburgh Parallel Computing Centre
- General Electric Company
- German National Research Center for Information Technology
- Hewlett-Packard
- Hitachi
- Hughes Aircraft Company
- Intel Corporation
- International Business Machines
- Khoral Research
- Lawrence Livermore National Laboratory
- Los Alamos National Laboratory
- MPI Software Technology, Inc.
- Mississippi State University
- NEC Corporation
- National Aeronautics and Space Administration
- National Energy Research Scientific Computing Center
- National Institute of Standards and Technology
- National Oceanic and Atmospheric Administration
- Oak Ridge National Laboratory
- Ohio State University
- PALLAS GmbH

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

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

- Argonne National Laboratory
- Bolt, Beranek, and Newman
- California Institute of Technology
- Center for Computing Sciences
- Convex Computer Corporation
- Cray Research
- Digital Equipment Corporation
- Dolphin Interconnect Solutions, Inc.
- Edinburgh Parallel Computing Centre
- General Electric Company
- German National Research Center for Information Technology
- Hewlett-Packard
- Hitachi
- Hughes Aircraft Company
- Intel Corporation
- International Business Machines
- Khoral Research
- Lawrence Livermore National Laboratory
- Los Alamos National Laboratory
- MPI Software Technology, Inc.
- Mississippi State University
- NEC Corporation
- National Aeronautics and Space Administration
- National Energy Research Scientific Computing Center
- National Institute of Standards and Technology
- National Oceanic and Atmospheric Administration
- Oak Ridge National Laboratory
- Ohio State University
- PALLAS GmbH
MPI-2 operated on a very tight budget (in reality, it had no budget when the first meeting was announced). Many institutions helped the MPI-2 effort by supporting the efforts and travel of the members of the MPI Forum. Direct support was given by NSF and DARPA under NSF contract CDA-9115428 for travel by U.S. academic participants and Esprit under project HPC Standards (21111) for European participants.

MPI-1.3 and MPI-2.1:
The editors and organizers of the combined documents have been:

- Richard Graham, Convener and Meeting Chair
- Jack Dongarra, Steering Committee
- Al Geist, Steering Committee
- Bill Gropp, Steering Committee
- Rainer Keller, Merge of MPI-1.3
- Andrew Lumsdaine, Steering Committee
- Ewing Lusk, Steering Committee, MPI-1.1-Errata (Oct. 12, 1998) MPI-2.1-Errata Ballots 1, 2 (May 15, 2002)
All chapters have been revisited to achieve a consistent MPI-2.1 text. Those who served as authors for the necessary modifications are:

- Bill Gropp, Frontmatter, Introduction, and Bibliography
- Richard Graham, Point-to-Point Communications
- Adam Moody, Collective Communication
- Richard Treumann, Groups, Contexts, and Communicators
- Jesper Larsson Träff, Process Topologies, Info-Object, and One-Sided Communications
- George Bosilca, Environmental Management
- David Solt, Process Creation and Management
- Bronis de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O
- Jeff Squyres, Language Bindings
- Rolf Rabenseifner, Deprecated Functions, and Annex Change-Log
- Alexander Supalov and Denis Nagomy, Annex Language bindings

The following list includes some of the active participants who attended MPI-2 Forum meetings and in the e-mail discussions of the errata items and are not mentioned above.

Pavan Balaji Purushotham V. Bangalore Brian Barrett Richard Barrett
Christian Bell Robert Blackmore Gil Bloch Ron Brightwell
Jeffrey Brown Darius Buntinas Jonathan Carter Nathan DeBardeleben
Terry Donatje Gabor Dona Edric Ellis Karl Feind
Edgar Gabriel Patrick Geoffray David Gingold Dave Goodell
Erez Haba Robert Harrison Thomas Herault Steve Hodson
Torsten Hoefler Joshua Hursey Yann Kalemkerian Matthew Koop
Quincey Koziol Sameer Kumar Miron Livny Kannan Narasimhan
Mark Pagel Avneesh Pant Steve Poole Howard Pritchard
Craig Rasmussen Hubert Ritzdorf Rob Ross Tony Skjellum
Brian Smith Vinod Tippaguraj Jesper Larsson Träff Keith Underwood

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

The following institutions supported the MPI-2 effort through time and travel support for the people listed above.

Argonne National Laboratory
Bull
In addition, the HDF Group provided travel support for one U.S. academic.
Chapter 1

Introduction to MPI

1.1 Overview and Goals

MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. (Extensions to the “classical” message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.) MPI is a specification, not an implementation; there are multiple implementations of MPI. This specification is for a library interface: MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, Fortran-77, and Fortran-95, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers. The next few sections provide an overview of the history of MPI’s development.

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

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

A complete list of goals follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
• Allow convenient C, C++, Fortran-77, and Fortran-95 bindings for the interface.

• Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.

• Define an interface that can be implemented on many vendor’s platforms, with no significant changes in the underlying communication and system software.

• Semantics of the interface should be language independent.

• The interface should be designed to allow for thread safety.

1.2 Background of MPI-1.0

MPI sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI was strongly influenced by work at the IBM T. J. Watson Research Center [1, 2], Intel’s NX/2 [8], Express [12], nCUBE’s Vertex [34], p4 [7, 8], and PARMACS [5, 9]. Other important contributions have come from Zipcode [40, 41], Chimp [16, 17], PVM [4, 14], Chameleon [25], and PICL [24].

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message-Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [48]. At this workshop the basic features essential to a standard message-passing interface were discussed, and a working group established to continue the standardization process.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [15]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and “get the ball rolling,” it focused mainly on point-to-point communications. MPI1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was decided to place the standardization process on a more formal footing, and to generally adopt the procedures and organization of the High Performance Fortran Forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discussion together constituted the MPI Forum, membership of which has been open to all members of the high performance computing community.
1.3 Background of MPI-1.1, MPI-1.2, and MPI-2.0

Beginning in March 1995, the MPI Forum began meeting to consider corrections and extensions to the original MPI Standard document [21]. The first product of these deliberations was Version 1.1 of the MPI specification, released in June of 1995 [22] (see \url{http://www.mpi-forum.org} for official document releases). At that time, efforts focused in five areas.

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

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

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

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

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

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

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

Restarting regular work of the MPI Forum was initiated in three meetings, at EuroPVM/MPI’06 in Bonn, at EuroPVM/MPI’07 in Paris, and at SC’07 in Reno. In December 2007, a steering committee started the organization of new MPI Forum meetings at regular 8-weeks intervals. At the January 14-16, 2008 meeting in Chicago, the MPI Forum decided to combine the existing and future MPI documents to one single document for each version of the MPI standard. For technical and historical reasons, this series was started with MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1-4) were combined into one draft document, for each chapter, a chapter author and review team were defined. They cleaned up the document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document was finished in June 2008, and finally released with a second vote in September 2008 in the meeting at Dublin, just before EuroPVM/MPI’08. The major work of the current MPI Forum is the preparation of MPI-3.

1.5 Who Should Use This Standard?

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

1.6 What Platforms Are Targets For Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those “machines” consisting of collections of other machines, parallel or not, connected by a communication network. The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogeneous networks of workstations.

MPI-2.0 was released, and a second ballot was voted on May 22, 2002. Both votes were done electronically. Both ballots were combined into one document: “Errata for MPI-2”, May 15, 2002. This errata process was then interrupted, but the Forum and its e-mail reflectors kept working on new requests for clarification.

Restarting regular work of the MPI Forum was initiated in three meetings, at EuroPVM/MPI’06 in Bonn, at EuroPVM/MPI’07 in Paris, and at SC’07 in Reno. In December 2007, a steering committee started the organization of new MPI Forum meetings at regular 8-weeks intervals. At the January 14-16, 2008 meeting in Chicago, the MPI Forum decided to combine the existing and future MPI documents to one single document for each version of the MPI standard. For technical and historical reasons, this series was started with MPI-1.3. Additional Ballots 3 and 4 solved old questions from the errata list started in 1995 up to new questions from the last years. After all documents (MPI-1.1, MPI-2, Errata for MPI-1.1 (Oct. 12, 1998), and MPI-2.1 Ballots 1-4) were combined into one draft document, for each chapter, a chapter author and review team were defined. They cleaned up the document to achieve a consistent MPI-2.1 document. The final MPI-2.1 standard document was finished in June 2008, and finally released with a second vote in September 2008 in the meeting at Dublin, just before EuroPVM/MPI’08. The major work of the current MPI Forum is the preparation of MPI-3.

1.5 Who Should Use This Standard?

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

1.6 What Platforms Are Targets For Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those “machines” consisting of collections of other machines, parallel or not, connected by a communication network. The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogeneous networks of workstations.
1.7 What Is Included In The Standard?

The standard includes:

- Point-to-point communication
- Datatypes
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Environmental Management and inquiry
- The info object
- Process creation and management
- One-sided communication
- External interfaces
- Parallel file I/O
- Language Bindings for Fortran, C and C++
- Profiling interface

1.8 What Is Not Included In The Standard?

The standard does not specify:

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

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

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

- Chapter 2, MPI Terms and Conventions, explains notational terms and conventions used throughout the MPI document.
- Chapter 3, Point to Point Communication, defines the basic, pairwise communication subset of MPI. Send and receive are found here, along with many associated functions designed to make basic communication powerful and efficient.
- Chapter 4, Datatypes, defines a method to describe any data layout, e.g., an array of structures in the memory, which can be used as message send or receive buffer.
- Chapter 5, Collective Communications, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations.
- Chapter 6, Groups, Contexts, Communicators, and Caching, shows how groups of processes are formed and manipulated, how unique communication contexts are obtained, and how the two are bound together into a communicator.
- Chapter 7, Process Topologies, explains a set of utility functions meant to assist in the mapping of process groups (a linearly ordered set) to richer topological structures such as multi-dimensional grids.
- Chapter 8, MPI Environmental Management, explains how the programmer can manage and make inquiries of the current MPI environment. These functions are needed for the writing of correct, robust programs, and are especially important for the construction of highly-portable message-passing programs.
- Chapter 9, The Info Object, defines an opaque object, that is used as input of several MPI routines.
- Chapter 10, Process Creation and Management, defines routines that allow for creation of processes.
- Chapter 11, One-Sided Communications, defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations.
- Chapter 12, External Interfaces, defines routines designed to allow developers to layer on top of MPI. This includes generalized requests, routines that decode MPI opaque objects, and threads.
- Chapter 13, I/O, defines MPI support for parallel I/O.
• Chapter 14, Profiling Interface, explains a simple name-shifting convention that any MPI implementation must support. One motivation for this is the ability to put performance profiling calls into MPI without the need for access to the MPI source code. The name shift is merely an interface, it says nothing about how the actual profiling should be done and in fact, the name shift can be useful for other purposes.

• Chapter 15, Deprecated Functions, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.

• Chapter 16, Language Bindings, describes the C++ binding, discusses Fortran issues, and describes language interoperability aspects between C, C++, and Fortran.

The Appendices are:

• Annex A, Language Bindings Summary, gives specific syntax in C, C++, and Fortran, for all MPI functions, constants, and types.

• Annex B, Change-Log, summarizes major changes since the previous version of the standard.

• Several Index pages are showing the locations of examples, constants and predefined routines, callback routines’ prototypes, and all MPI functions.

MPI provides various interfaces to facilitate interoperability of distinct MPI implementations. Among these are the canonical data representation for MPI I/O and for MPI_PACK_EXTERNAL and MPI_UNPACK_EXTERNAL. The definition of an actual binding of these interfaces that will enable interoperability is outside the scope of this document.

A separate document consists of ideas that were discussed in the MPI Forum and deemed to have value, but are not included in the MPI Standard. They are part of the “Journal of Development” (JOD), lest good ideas be lost and in order to provide a starting point for further work. The chapters in the JOD are

• Chapter 2, Spawning Independent Processes, includes some elements of dynamic process management, in particular management of processes with which the spawning processes do not intend to communicate, that the Forum discussed at length but ultimately decided not to include in the MPI Standard.

• Chapter 3, Threads and MPI, describes some of the expected interaction between an MPI implementation and a thread library in a multi-threaded environment.

• Chapter 4, Communicator ID, describes an approach to providing identifiers for communicators.

• Chapter 5, Miscellany, discusses Miscellaneous topics in the MPI JOD, in particular single-copy routines for use in shared-memory environments and new datatype constructors.

• Chapter 6, Toward a Full Fortran 90 Interface, describes an approach to providing a more elaborate Fortran 90 interface.
- Chapter 7, Split Collective Communication, describes a specification for certain non-blocking collective operations.
- Chapter 8, Real-Time MPI, discusses MPI support for real time processing.
Chapter 2

MPI Terms and Conventions

This chapter explains notational terms and conventions used throughout the MPI document, some of the choices that have been made, and the rationale behind those choices. It is similar to the MPI-1 Terms and Conventions chapter but differs in some major and minor ways. Some of the major areas of difference are the naming conventions, some semantic definitions, file objects, Fortran 90 vs Fortran 77, C++, processes, and interaction with signals.

2.1 Document Notation

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

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

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

2.2 Naming Conventions

In many cases MPI names for C functions are of the form Class\_action\_subset. This convention originated with MPI-1. Since MPI-2 an attempt has been made to standardize the names of MPI functions according to the following rules. The C++ bindings in particular follow these rules (see Section 2.6.4 on page 18).

1. In C, all routines associated with a particular type of MPI object should be of the form Class\_action\_subset or, if no subset exists, of the form Class\_action. In Fortran, all routines associated with a particular type of MPI object should be of the form CLASS\_ACTION\_SUBSET or, if no subset exists, of the form CLASS\_ACTION. For C
and Fortran we use the C++ terminology to define the Class. In C++, the routine is a method on `Class` and is named `MPI::Class::Action_subset`. If the routine is associated with a certain class, but does not make sense as an object method, it is a static member function of the class.

2. If the routine is not associated with a class, the name should be of the form `Action_subset` in C and `ACTION_SUBSET` in Fortran, and in C++ should be scoped in the `MPI` namespace, `MPI::Action_subset`.

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

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

MPI identifiers are limited to 30 characters (31 with the profiling interface). This is done to avoid exceeding the limit on some compilation systems.

2.3 Procedure Specification

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

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

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

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

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

A common occurrence for MPI functions is an argument that is used as `IN` by some processes and `OUT` by other processes. Such an argument is, syntactically, an
A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.

2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

nonblocking A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is started by the call that initiates it, e.g., MPI_ISEND. The word complete is used with respect to operations, requests, and communications. An operation completes when the user is allowed to reuse resources, and any output buffers have been updated; i.e. a call to MPI_TEST will return flag = true. A request is completed by a call to wait, which returns, or a test or get status call which returns flag = true. This completing call has two effects: the status is extracted from the request; in the case of test and wait, if the request was nonpersistent, it is freed, and becomes inactive if it was persistent. A communication completes when all participating operations complete.

blocking A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.

local A procedure is local if completion of the procedure depends only on the local executing process.

non-local A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process.
collective A procedure is collective if all processes in a process group need to invoke the procedure. A collective call may or may not be synchronizing. Collective calls over the same communicator must be executed in the same order by all members of the process group.

predefined A predefined datatype is a datatype with a predefined (constant) name (such as MPI_INT, MPI_FLOAT, or MPI_LONG) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or MPI_TYPE_CREATE_F90_COMPLEX. The former are named whereas the latter are unnamed.

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

portable A datatype is portable, if it is a predefined datatype, or it is derived from a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable because all displacements in the datatype are in terms of extents of one predefined datatype. Therefore, if such a datatype fits a data layout in one memory, it will fit the corresponding data layout in another memory, if the same declarations were used, even if the two systems have different architectures. On the other hand, if a datatype was constructed using MPI_TYPE_CREATE_INDEXED, MPI_TYPE_CREATE_HVECTOR or MPI_TYPE_CREATE_STRUCT, then the datatype contains explicit byte displacements (e.g., providing padding to meet alignment restrictions). These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are used for data layouts on another process, running on a processor with a different architecture.

equivalent Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names.

2.5 Data Types

2.5.1 Opaque Objects

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

In Fortran, all handles have type INTEGER. In C and C++, a different handle type is defined for each category of objects. In addition, handles themselves are distinct objects in C++. The C and C++ types must support the use of the assignment and equality operators.
Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. C++ handles can simply “wrap up” a table index or pointer.

(End of advice to implementors.)

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the objects are described. The calls accept a handle argument of matching type. In an allocate call this is an OUT argument that returns a valid reference to the object. In a call to deallocate this is an INOUT argument which returns with an “invalid handle” value. MPI provides an “invalid handle” constant for each object type. Comparisons to this constant are used to test for validity of the handle.

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

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

MPI provides certain predefined opaque objects and predefined, static handles to these objects. The user must not free such objects. In C++, this is enforced by declaring the handles to these predefined objects to be static const.

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

The explicit separation of handles in user space and objects in system space allows space-reclaiming and deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed.

The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. (End of rationale.)

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

Advice to implementors. In Fortran, the handle can be an index into a table of opaque objects in a system table; in C it can be such an index or a pointer to the object. C++ handles can simply “wrap up” a table index or pointer.

(End of advice to implementors.)

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the objects are described. The calls accept a handle argument of matching type. In an allocate call this is an OUT argument that returns a valid reference to the object. In a call to deallocate this is an INOUT argument which returns with an “invalid handle” value. MPI provides an “invalid handle” constant for each object type. Comparisons to this constant are used to test for validity of the handle.

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

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

MPI provides certain predefined opaque objects and predefined, static handles to these objects. The user must not free such objects. In C++, this is enforced by declaring the handles to these predefined objects to be static const.

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

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

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. The alternative would have been to allow handles to have been an arbitrary, opaque type. This would force the introduction of routines to do assignment and comparison, adding complexity, and was therefore ruled out. (End of rationale.)

Advice to users. A user may accidently create a dangling reference by assigning to a handle the value of another handle, and then deallocating the object associated with these handles. Conversely, if a handle variable is deallocated before the associated object is freed, then the object becomes inaccessible (this may occur, for example, if the handle is a local variable within a subroutine, and the subroutine is exited before the associated object is deallocated). It is the user’s responsibility to avoid adding or deleting references to opaque objects, except as a result of MPI calls that allocate or deallocate such objects. (End of advice to users.)
Advice to implementors. The intended semantics of opaque objects is that opaque objects are separate from one another; each call to allocate such an object copies all the information required for the object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather than copies of its components; a call to MPI_COMM_GROUP may return a reference to the group associated with the communicator, rather than a copy of this group. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects in such a way that the visible effect is as if the objects were copied. (End of advice to implementors.)

2.5.2 Array Arguments

An MPI call may need an argument that is an array of opaque objects, or an array of handles. The array-of-handles is a regular array with entries that are handles to objects of the same type in consecutive locations in the array. Whenever such an array is used, an additional len argument is required to indicate the number of valid entries (unless this number can be derived otherwise). The valid entries are at the beginning of the array; len indicates how many of them there are, and need not be the size of the entire array.

The same approach is followed for other array arguments. In some cases NULL handles are considered valid entries. When a NULL argument is desired for an array of statuses, one uses MPI_STATUSES_IGNORE.

2.5.3 State

MPI procedures use at various places arguments with state types. The values of such a data type are all identified by names, and no operation is defined on them. For example, the MPI_TYPE_CREATE_SUBARRAY routine has a state argument order with values MPI_ORDER_C and MPI_ORDER_FORTRAN.

2.5.4 Named Constants

MPI procedures sometimes assign a special meaning to a special value of a basic type argument; e.g., tag is an integer-valued argument of point-to-point communication operations, with a special wild-card value, MPI_ANY_TAG. Such arguments will have a range of regular values, which is a proper subrange of the range of values of the corresponding basic type; special values (such as MPI_ANY_TAG) will be outside the regular range. The range of regular values, such as tag, can be queried using environmental inquiry functions (Chapter 7 of the MPI-1 document). The range of other values, such as source, depends on values given by other MPI routines (in the case of source it is the communicator size).

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE).

The constants that cannot be used in initialization expressions or assignments in Fortran are:

```
MPI_BOTTOM
MPI_STATUS_IGNORE
```
2.5.5 Choice

MPI functions sometimes use arguments with a choice (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran, the document uses $<\text{type}>$ to represent a choice variable; for C and C++, we use void *.

2.5.6 Addresses

Some MPI procedures use address arguments that represent an absolute address in the calling program. The datatype of such an argument is MPI_Aint in C; MPI::Aint in C++ and INTEGER (KIND=MPI_ADDRESS_KIND) in Fortran. There is the MPI constant MPI_BOTTOM to indicate the start of the address range.

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities can easily be larger than 32 bits which can be the default size of a Fortran integer. To overcome this, these quantities are declared to be INTEGER (KIND=MPI_Offset_KIND) in Fortran. In C one uses MPI_Offset whereas in C++ one uses MPI::Offset.

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, ISO C, and C++, in particular. (Note that ANSI C has been replaced by ISO C.) Defined here are various object representations, as well as the naming conventions used for expressing this standard. Actual calling sequences are defined elsewhere. MPI bindings are for Fortran 90, though they are designed to be usable in Fortran 77 environments.

Since the word PARAMETER is a keyword in the Fortran language, we use the word “argument” to denote the arguments to a subroutine. These are normally referred to as parameters in C and C++, however, we expect that C and C++ programmers will use different mechanisms for providing such arguments will differ from language to language. For Fortran, the document uses $<\text{type}>$ to represent a choice variable; for C and C++, we use void *.

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through parameter statements) is not possible because an implementation cannot distinguish these values from 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). (End of advice to implementors.)

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through parameter statements) is not possible because an implementation cannot distinguish these values from 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). (End of advice to implementors.)

2.6. LANGUAGE BINDING

MPI functions sometimes use arguments with a choice (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran, the document uses $<\text{type}>$ to represent a choice variable; for C and C++, we use void *.

2.5.6 Addresses

Some MPI procedures use address arguments that represent an absolute address in the calling program. The datatype of such an argument is MPI_Aint in C; MPI::Aint in C++ and INTEGER (KIND=MPI_ADDRESS_KIND) in Fortran. There is the MPI constant MPI_BOTTOM to indicate the start of the address range.

2.5.7 File Offsets

For I/O there is a need to give the size, displacement, and offset into a file. These quantities can easily be larger than 32 bits which can be the default size of a Fortran integer. To overcome this, these quantities are declared to be INTEGER (KIND=MPI_Offset_KIND) in Fortran. In C one uses MPI_Offset whereas in C++ one uses MPI::Offset.

2.6 Language Binding

This section defines the rules for MPI language binding in general and for Fortran, ISO C, and C++, in particular. (Note that ANSI C has been replaced by ISO C.) Defined here are various object representations, as well as the naming conventions used for expressing this standard. Actual calling sequences are defined elsewhere. MPI bindings are for Fortran 90, though they are designed to be usable in Fortran 77 environments.

Since the word PARAMETER is a keyword in the Fortran language, we use the word “argument” to denote the arguments to a subroutine. These are normally referred to as parameters in C and C++, however, we expect that C and C++ programmers will use different mechanisms for providing such arguments will differ from language to language. For Fortran, the document uses $<\text{type}>$ to represent a choice variable; for C and C++, we use void *.

Advice to implementors. In Fortran the implementation of these special constants may require the use of language constructs that are outside the Fortran standard. Using special values for the constants (e.g., by defining them through parameter statements) is not possible because an implementation cannot distinguish these values from 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). (End of advice to implementors.)
understand the word “argument” (which has no specific meaning in C/C++), thus allowing us to avoid unnecessary confusion for Fortran programmers.

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid the “mpi_” and “pmpi_” prefixes.

### 2.6.1 Deprecated Names and Functions

A number of chapters refer to deprecated or replaced MPI-1 constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users are recommended not to continue using, since better solutions were provided with MPI-2.

For example, the Fortran binding for MPI-1 functions that have address arguments uses integer. This is not consistent with the C binding, and causes problems on machines with 32 bit integers and 64 bit addresses. In MPI-2, these functions were given new names with new bindings for the address arguments. The use of the old functions is deprecated. For consistency, here and in a few other cases, new C functions are also provided, even though the new functions are equivalent to the old functions. The old names are deprecated.

Another example is provided by the MPI-1 predefined datatypes MPI_USB and MPI_LB. They are deprecated, since their use is awkward and error-prone. The MPI-2 function MPI_TYPE_CREATE_RESIZED provides a more convenient mechanism to achieve the same effect.

Table 2.1 shows a list of all of the deprecated constructs. Note that the constants MPI_LB and MPI_USB are replaced by the function MPI_TYPE_CREATE_RESIZED; this is because their principal use was as input datatypes to MPI_TYPE_STRUCT to create resized datatypes. Also note that some Ctypedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

### 2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term Fortran is used it means Fortran 90.

All MPI names have an MPI_ prefix, and all characters are capitals. Programs must not declare variables, parameters, or functions with names beginning with the prefix MPI_. To avoid conflicting with the profiling interface, programs should also avoid functions with the prefix PMPI_. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than in C and C++ as discussed in Section 16.3.9. Handles are represented in Fortran as INTEGERs. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

The MPI Fortran binding is inconsistent with the Fortran 90 standard in several respects. These inconsistencies, such as register optimization problems, have implications for

Since Fortran is case insensitive, linkers may use either lower case or upper case when resolving Fortran names. Users of case sensitive languages should avoid the “mpi_” and “pmpi_” prefixes.

### 2.6.1 Deprecated Names and Functions

A number of chapters refer to deprecated or replaced MPI-1 constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users are recommended not to continue using, since better solutions were provided with MPI-2.

For example, the Fortran binding for MPI-1 functions that have address arguments uses integer. This is not consistent with the C binding, and causes problems on machines with 32 bit integers and 64 bit addresses. In MPI-2, these functions were given new names with new bindings for the address arguments. The use of the old functions is deprecated. For consistency, here and in a few other cases, new C functions are also provided, even though the new functions are equivalent to the old functions. The old names are deprecated.

Another example is provided by the MPI-1 predefined datatypes MPI_USB and MPI_LB. They are deprecated, since their use is awkward and error-prone. The MPI-2 function MPI_TYPE_CREATE_RESIZED provides a more convenient mechanism to achieve the same effect.

Table 2.1 shows a list of all of the deprecated constructs. Note that the constants MPI_LB and MPI_USB are replaced by the function MPI_TYPE_CREATE_RESIZED; this is because their principal use was as input datatypes to MPI_TYPE_STRUCT to create resized datatypes. Also note that some Ctypedefs and Fortran subroutine names are included in this list; they are the types of callback functions.

### 2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term Fortran is used it means Fortran 90.

All MPI names have an MPI_ prefix, and all characters are capitals. Programs must not declare variables, parameters, or functions with names beginning with the prefix MPI_. To avoid conflicting with the profiling interface, programs should also avoid functions with the prefix PMPI_. This is mandated to avoid possible name collisions.

All MPI Fortran subroutines have a return code in the last argument. A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than in C and C++ as discussed in Section 16.3.9. Handles are represented in Fortran as INTEGERs. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

The MPI Fortran binding is inconsistent with the Fortran 90 standard in several respects. These inconsistencies, such as register optimization problems, have implications for
An MPI subroutine with a choice argument may be called with different argument types.

An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument.

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.

An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program executing outside MPI calls.

Several named “constants,” such as MPI_BOTTOM, MPI_STATUS_IGNORE, and MPI_ERRCODES_IGNORE, are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 on page 14 for more information.

### Table 2.1: Deprecated constructs

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>MPI-2 Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ADDRESS</td>
<td>MPI_GET_ADDRESS</td>
</tr>
<tr>
<td>MPI_TYPE_HINDEXED</td>
<td>MPI_TYPE_CREATE_HINDEXED</td>
</tr>
<tr>
<td>MPI_TYPE_HVECTOR</td>
<td>MPI_TYPE_CREATE_HVECTOR</td>
</tr>
<tr>
<td>MPI_TYPE_STRUCT</td>
<td>MPI_TYPE_CREATE_STRUCT</td>
</tr>
<tr>
<td>MPI_TYPE_EXTENT</td>
<td>MPI_TYPE_GET_EXTENT</td>
</tr>
<tr>
<td>MPI_TYPE_UB</td>
<td>MPI_TYPE_GET_UB</td>
</tr>
<tr>
<td>MPI_TYPE_LB</td>
<td>MPI_TYPE_CREATE_RESIZED</td>
</tr>
<tr>
<td>MPI_TYPE_BOTTOM</td>
<td>MPI_TYPE_CREATE_BOTTOM</td>
</tr>
<tr>
<td>MPI_TYPE_STATUS</td>
<td>MPI_TYPE_CREATE_STATUS</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_CREATE</td>
<td>MPI_COMM_CREATE_ERRHANDLER</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_GET</td>
<td>MPI_COMM_ERRHANDLER</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_SET</td>
<td>MPI_COMM_SET_ERRHANDLER</td>
</tr>
<tr>
<td>MPI_Handler_function</td>
<td>MPI_Comm_errhandler_fn</td>
</tr>
<tr>
<td>MPI_KEYVAL_CREATE</td>
<td>MPI_COMM_CREATE_KEYVAL</td>
</tr>
<tr>
<td>MPI_KEYVAL_FREE</td>
<td>MPI_COMM_FREE_KEYVAL</td>
</tr>
<tr>
<td>MPI_DUP_FN</td>
<td>MPI_COMM_DUP_FN</td>
</tr>
<tr>
<td>MPI_NULL_COPY_FN</td>
<td>MPI_COMM_NULL_COPY_FN</td>
</tr>
<tr>
<td>MPI_NULL_DELETE_FN</td>
<td>MPI_COMM_NULL_DELETE_FN</td>
</tr>
<tr>
<td>MPI_Copy_function</td>
<td>MPI_Comm_copy_attr_fn</td>
</tr>
<tr>
<td>COPY_FUNCTION</td>
<td>COMM_COPY_ATTR_FN</td>
</tr>
<tr>
<td>MPIDelete_function</td>
<td>MPI_Comm_delete_attr_fn</td>
</tr>
<tr>
<td>DELETE_FUNCTION</td>
<td>COMM_DELETE_ATTR_FN</td>
</tr>
<tr>
<td>MPIAttrDELETE</td>
<td>MPI_COMM_DELETE_ATTR</td>
</tr>
<tr>
<td>MPI_ATTR_GET</td>
<td>MPI_COMM_GET_ATTR</td>
</tr>
<tr>
<td>MPI_ATTR_PUT</td>
<td>MPI_COMM_SET_ATTR</td>
</tr>
</tbody>
</table>

user codes that are discussed in detail in Section 16.2.2. They are also inconsistent with Fortran 77.

### Table 2.1: Deprecated constructs

<table>
<thead>
<tr>
<th>Deprecated</th>
<th>MPI-2 Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ADDRESS</td>
<td>MPI_GET_ADDRESS</td>
</tr>
<tr>
<td>MPI_TYPE_HINDEXED</td>
<td>MPI_TYPE_CREATE_HINDEXED</td>
</tr>
<tr>
<td>MPI_TYPE_HVECTOR</td>
<td>MPI_TYPE_CREATE_HVECTOR</td>
</tr>
<tr>
<td>MPI_TYPE_STRUCT</td>
<td>MPI_TYPE_CREATE_STRUCT</td>
</tr>
<tr>
<td>MPI_TYPE_EXTENT</td>
<td>MPI_TYPE_GET_EXTENT</td>
</tr>
<tr>
<td>MPI_TYPE_UB</td>
<td>MPI_TYPE_GET_UB</td>
</tr>
<tr>
<td>MPI_TYPE_LB</td>
<td>MPI_TYPE_CREATE_RESIZED</td>
</tr>
<tr>
<td>MPI_TYPE_BOTTOM</td>
<td>MPI_TYPE_CREATE_BOTTOM</td>
</tr>
<tr>
<td>MPI_TYPE_STATUS</td>
<td>MPI_TYPE_CREATE_STATUS</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_CREATE</td>
<td>MPI_COMM_CREATE_ERRHANDLER</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_GET</td>
<td>MPI_COMM_ERRHANDLER</td>
</tr>
<tr>
<td>MPI_ERRHANDLER_SET</td>
<td>MPI_COMM_SET_ERRHANDLER</td>
</tr>
<tr>
<td>MPI_Handler_function</td>
<td>MPI_Comm_errhandler_fn</td>
</tr>
<tr>
<td>MPI_KEYVAL_CREATE</td>
<td>MPI_COMM_CREATE_KEYVAL</td>
</tr>
<tr>
<td>MPI_KEYVAL_FREE</td>
<td>MPI_COMM_FREE_KEYVAL</td>
</tr>
<tr>
<td>MPI_DUP_FN</td>
<td>MPI_COMM_DUP_FN</td>
</tr>
<tr>
<td>MPI_NULL_COPY_FN</td>
<td>MPI_COMM_NULL_COPY_FN</td>
</tr>
<tr>
<td>MPI_NULL_DELETE_FN</td>
<td>MPI_COMM_NULL_DELETE_FN</td>
</tr>
<tr>
<td>MPI_Copy_function</td>
<td>MPI_Comm_copy_attr_fn</td>
</tr>
<tr>
<td>COPY_FUNCTION</td>
<td>COMM_COPY_ATTR_FN</td>
</tr>
<tr>
<td>MPIDelete_function</td>
<td>MPI_Comm_delete_attr_fn</td>
</tr>
<tr>
<td>DELETE_FUNCTION</td>
<td>COMM_DELETE_ATTR_FN</td>
</tr>
<tr>
<td>MPIAttrDELETE</td>
<td>MPI_COMM_DELETE_ATTR</td>
</tr>
<tr>
<td>MPI_ATTR_GET</td>
<td>MPI_COMM_GET_ATTR</td>
</tr>
<tr>
<td>MPI_ATTR_PUT</td>
<td>MPI_COMM_SET_ATTR</td>
</tr>
</tbody>
</table>

user codes that are discussed in detail in Section 16.2.2. They are also inconsistent with Fortran 77.
Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, `mpif.h`. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., `MPI_ADDRESS_KIND` and `MPI_OFFSET_KIND`) that hold address information. On systems that do not support Fortran 90-style parameterized types, `INTEGER*8` or `INTEGER` should be used instead.
- The memory allocation routine `MPI_ALLOC_MEM` cannot be usefully used in Fortran without a language extension that allows the allocated memory to be associated with a Fortran variable.

2.6.3 C Binding Issues

We use the ISO C declaration format. All MPI names have an `MPI_` prefix, defined constants are in all capital letters, and defined types and functions have one capital letter after the prefix. Programs must not declare variables or functions with names beginning with the prefix `MPI_`. To support the profiling interface, programs should not declare functions with names beginning with the prefix `PMPI_`.

The definition of named constants, function prototypes, and type definitions must be supplied in an include file `mpif.h`.

Almost all C functions return an error code. The successful return code will be `MPI_SUCCESS`, but failure return codes are implementation dependent.

Type declarations are provided for handles to each category of opaque objects. Array arguments are indexed from zero. Logical flags are integers with value 0 meaning “false” and a non-zero value meaning “true.” Choice arguments are pointers of type `void *`. Address arguments are of MPI defined type `MPI_Aint`. File displacements are of type `MPI_Offset`. `MPI_Aint` is defined to be an integer of the size needed to hold any valid address on the target architecture. `MPI_Offset` is defined to be an integer of the size needed to hold any valid file size on the target architecture.

2.6.4 C++ Binding Issues

There are places in the standard that give rules for C and not for C++. In these cases, the C rule should be applied to the C++ case, as appropriate. In particular, the values of constants given in the text are the ones for C and Fortran. A cross index of these with the C++ names is given in Annex A.

We use the ISO C++ declaration format. All MPI names are declared within the scope of a namespace called `MPI` and therefore are referenced with an `MPI:::` prefix. Defined constants are in all capital letters, and class names, defined types, and functions have only their first letter capitalized. Programs must not declare variables or functions in the `MPI` namespace. This is mandated to avoid possible name collisions.
Advice to implementors.  The file mpi.h may contain both the C and C++ definitions.  Usually one can simply use the defined value (generally _cplamplus, but not required) to see if one is using C++ to protect the C++ definitions.  It is possible that a C compiler will require that the source protected this way be legal C code.  In this case, all the C++ definitions can be placed in a different include file and the "#include" directive can be used to include the necessary C++ definitions in the mpi.h file.  (End of advice to implementors.)

C++ functions that create objects or return information usually place the object or information in the return value. Since the language neutral prototypes of MPI functions include the C++ return value as an OUT parameter, semantic descriptions of MPI functions refer to the C++ return value by that parameter name. The remaining C++ functions return void.

In some circumstances, MPI permits users to indicate that they do not want a return value.  For example, the user may indicate that the status is not filled in.  Unlike C and Fortran where this is achieved through a special input value, in C++ this is done by having two bindings where one has the optional argument and one does not.

C++ functions do not return error codes.  If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an MPI::Exception object.

It should be noted that the default error handler (i.e., MPI::ERRORS_ARE_FATAL) on a given type has not changed. User error handlers are also permitted.  MPI::ERRORS_RETURN simply returns control to the calling function; there is no provision for the user to retrieve the error code.

User callback functions that return integer error codes should not throw exceptions; the returned error will be handled by the MPI implementation by invoking the appropriate error handler.

Advice to users.  C++ programmers that want to handle MPI errors on their own should use the MPI::ERRORS_THROW_EXCEPTIONS error handler, rather than MPI::ERRORS_RETURN, that is used for that purpose in C.  Care should be taken using exceptions in mixed language situations.  (End of advice to users.)

Opaque object handles must be objects in themselves, and have the assignment and equality operators overridden to perform semantically like their C and Fortran counterparts.

Array arguments are indexed from zero.

Logical flags are of type bool.

Choice arguments are pointers of type void *.

Address arguments are of MPI-defined integer type MPI::Aint, defined to be an integer of the size needed to hold any valid address on the target architecture. Analogously, MPI::Offset is an integer to hold file offsets.

Most MPI functions are methods of MPI C++ classes.  MPI class names are generated from the language neutral MPI types by dropping the MPI_ prefix and scoping the type within the MPI namespace.  For example, MPI_DATATYPE becomes MPI::Datatype.

The names of MPI functions generally follow the naming rules given.  In some circumstances, the MPI function is related to a function defined already for MPI-1 with a name
that does not follow the naming conventions. In this circumstance, the language neutral name is in analogy to the MPI name even though this gives an MPI-2 name that violates the naming conventions. The C and Fortran names are the same as the language neutral name in this case. However, the C++ names do reflect the naming rules and can differ from the C and Fortran names. Thus, the analogous name in C++ to the MPI name may be different than the language neutral name. This results in the C++ name differing from the language neutral name. An example of this is the language neutral name of MPI_FINALIZED and a C++ name of MPI::Is_finalized.

In C++, function typedefs are made publicly within appropriate classes. However, these declarations then become somewhat cumbersome, as with the following:

typedef MPI::Grequest::Query_function();

would look like the following:

```cpp
namespace MPI {
  class Request {
    // ...
  };

  class Grequest : public MPI::Request {
    // ...
    typedef Query_function(void* extra_state, MPI::Status& status);
  };
}
```

Rather than including this scaffolding when declaring C++ typedefs, we use an abbreviated form. In particular, we explicitly indicate the class and namespace scope for the typedef of the function. Thus, the example above is shown in the text as follows:

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

The C++ bindings presented in Annex A.4 and throughout this document were generated by applying a simple set of name generation rules to the MPI function specifications. While these guidelines may be sufficient in most cases, they may not be suitable for all situations. In cases of ambiguity or where a specific semantic statement is desired, these guidelines may be superseded as the situation dictates.

1. All functions, types, and constants are declared within the scope of a `namespace` called MPI.
2. Arrays of MPI handles are always left in the argument list (whether they are IN or OUT arguments).
3. If the argument list of an MPI function contains a scalar IN handle, and it makes sense to define the function as a method of the object corresponding to that handle, the function is made a member function of the corresponding MPI class. The member functions are named according to the corresponding MPI function name, but without the "MPI_" prefix and without the object name prefix (if applicable). In addition:
4. MPI functions are made into class functions (static) when they belong on a class but do not have a unique scalar IN or INOUT parameter of that class.

5. If the argument list contains a single OUT argument that is not of type MPI_STATUS (or an array), that argument is dropped from the list and the function returns that value.

Example 2.1 The C++ binding for MPI_COMM_SIZE is int MPI::Comm::Get_size(void) const.

6. If there are multiple OUT arguments in the argument list, one is chosen as the return value and is removed from the list.

7. If the argument list does not contain any OUT arguments, the function returns void.

Example 2.2 The C++ binding for MPI_REQUEST_FREE is void MPI::Request::Free(void)

8. MPI functions to which the above rules do not apply are not members of any class, but are defined in the MPI namespace.

Example 2.3 The C++ binding for MPI_BUFFER_ATTACH is void MPI::Attach_buffer(void* buffer, int size).

9. All class names, defined types, and function names have only their first letter capitalized. Defined constants are in all capital letters.

10. Any IN pointer, reference, or array argument must be declared const.

11. Handles are passed by reference.

12. Array arguments are denoted with square brackets ([]), not pointers, as this is more semantically precise.

2.6.5 Functions and Macros

An implementation is allowed to implement MPI_WTIME, MPI_WTICK, PMPI_WTIME, PMPI_WTICK, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 16.3.4, and no others, as macros in C.

Advice to implementors. Implementors should document which routines are implemented as macros. (End of advice to implementors.)

Advice to users. If these routines are implemented as macros, they will not work with the MPI profiling interface. (End of advice to users.)
CHAPTER 2. MPI TERMS AND CONVENTIONS

2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in a MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible.

This document specifies the behavior of a parallel program assuming that only MPI calls are used. The interaction of an MPI program with other possible means of communication, I/O, and process management is not specified. Unless otherwise stated in the specification of the standard, MPI places no requirements on the result of its interaction with external mechanisms that provide similar or equivalent functionality. This includes, but is not limited to, interactions with external mechanisms for process control, shared and remote memory access, file system access and control, interprocess communication, process signaling, and terminal I/O. High quality implementations should strive to make the results of such interactions intuitive to users, and attempt to document restrictions where deemed necessary.

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

The interaction of MPI and threads is defined in Section 12.4.

2.8 Error Handling

MPI provides the user with reliable message transmission. A message sent is always received correctly, and the user does not need to check for transmission errors, time-outs, or other error conditions. In other words, MPI does not provide mechanisms for dealing with failures in the communication system. If the MPI implementation is built on an unreliable underlying mechanism, then it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.

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

A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents. In C and Fortran, almost all MPI calls return a code that indicates successful completion of the operation. Whenever possible, MPI calls return an error code if an error occurred during the call. By default, an error detected during the execution of the MPI library causes the parallel computation to abort, except for file operations. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. The user may specify that no error is fatal, and handle error codes returned by MPI calls by himself.
or herself. Also, the user may provide his or her own error-handling routines, which will be invoked whenever an MPI call returns abnormally. The MPI error handling facilities are described in Section 8.3. The return values of C++ functions are not error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an MPI::Exception object. See also Section 16.1.8 on page 457.

Several factors limit the ability of MPI calls to return with meaningful error codes when an error occurs. MPI may not be able to detect some errors; other errors may be too expensive to detect in normal execution mode; finally some errors may be "catastrophic" and may prevent MPI from returning control to the caller in a consistent state.

Another subtle issue arises because of the nature of asynchronous communications: MPI calls may initiate operations that continue asynchronously after the call returned. Thus, the operation may return with a code indicating successful completion, yet later cause an error exception to be raised. If there is a subsequent call that relates to the same operation (e.g., a call that verifies that an asynchronous operation has completed) then the error argument associated with this call will be used to indicate the nature of the error. In a few cases, the error may occur after all calls that relate to the operation have completed, so that no error value can be used to indicate the nature of the error (e.g., an error on the receiver in a send with the ready mode). Such an error must be treated as fatal, since information cannot be returned for the user to recover from it.

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

Implementations may go beyond this document in supporting in a meaningful manner MPI calls that are defined here to be erroneous. For example, MPI specifies strict type matching rules between matching send and receive operations: it is erroneous to send a floating point variable and receive an integer. Implementations may go beyond these type matching rules, and provide automatic type conversion in such situations. It will be helpful to generate warnings for such non-conforming behavior.

MPI defines a way for users to create new error codes as defined in Section 8.5.

2.9 Implementation Issues

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

2.9.1 Independence of Basic Runtime Routines

MPI programs require that library routines that are part of the basic language environment (such as write in Fortran and printf and malloc in ISO C) and are executed after MPI_INIT and before MPI_FINALIZE operate independently and that their completion is independent of the action of other processes in an MPI program.
Note that this in no way prevents the creation of library routines that provide parallel services whose operation is collective. However, the following program is expected to complete in an ISO C environment regardless of the size of MPI_COMM_WORLD (assuming that printf is available at the executing nodes).

```c
int rank;
MPI_Init((void *)0, (void *)0);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) printf("Starting program\n");
MPI_Finalize();
```

The corresponding Fortran and C++ programs are also expected to complete.

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

```c
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("Output from task rank %d\n", rank);
```

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

2.9.2 Interaction with Signals

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

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

2.10 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the standard. Furthermore, the examples have not been carefully checked or verified.
Chapter 3

Point-to-Point Communication

3.1 Introduction

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

```c
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0) /* code for process zero */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank == 1) /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
        
        MPI_Finalize();
    }
}
```

In this example, process zero (`myrank = 0`) sends a message to process one using the `send` operation `MPI_SEND`. The operation specifies a `send buffer` in the sender memory from which the message data is taken. In the example above, the send buffer consists of the storage containing the variable `message` in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition,
3.2 Blocking Send and Receive Operations

3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

\[
\text{MPI\_SEND}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm})
\]

- \textbf{IN} buf initial address of send buffer (choice)
- \textbf{IN} count number of elements in send buffer (nonnegative integer)
- \textbf{IN} datatype datatype of each send buffer element (handle)
- \textbf{IN} dest rank of destination (integer)
- \textbf{IN} tag message tag (integer)
- \textbf{IN} comm communicator (handle)

\[
\text{int MPI\_Send(void* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)}
\]

The blocking semantics of this call are described in Section 3.4.
3.2. BLOCKING SEND AND RECEIVE OPERATIONS

3.2.2 Message Data

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

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

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

<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>_BYTE</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>_PACKED</td>
</tr>
</tbody>
</table>

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

The datatypes _BYTE and _PACKED do not correspond to a Fortran or C datatype. A value of type _BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type _PACKED is explained in Section 4.2.

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

<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>_BYTE</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>_PACKED</td>
</tr>
</tbody>
</table>

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

Possible values for this argument for C and the corresponding C types are listed in Table 3.2.

The datatypes _BYTE and _PACKED do not correspond to a Fortran or C datatype. A value of type _BYTE consists of a byte (8 binary digits). A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. The use of the type _PACKED is explained in Section 4.2.

Rationale. One goal of the design is to allow for MPI to be implemented as a library, with no need for additional preprocessing or compilation. Thus, one cannot assume that a communication call has information on the datatype of variables in the communication buffer; this information must be supplied by an explicit argument. The need for such datatype information will become clear in Section 3.3.2. (End of rationale.)
### Table 3.2: Predefined MPI datatypes corresponding to C datatypes

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG (as a synonym)</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>(treated as integral value)</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>(treated as integral value)</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_WCHAR</td>
<td>wchar_t</td>
</tr>
<tr>
<td>(defined in <code>&lt;stddef.h&gt;</code></td>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.2.3 Message Envelope

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

- **source**
- **destination**
- **tag**
- **communicator**

The message source is implicitly determined by the identity of the message sender. The other fields are specified by arguments in the send operation.

The message destination is specified by the **dest** argument.

The integer-valued message tag is specified by the **tag** argument. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. It can be found by querying the value of the attribute MPI_TAG_UB, as described in Chapter 8. MPI requires that UB be no less than 32767.

The **comm** argument specifies the **communicator** that is used for the send operation. Communicators are explained in Chapter 6; below is a brief summary of their usage.

Communicators are explained in Chapter 6; below is a brief summary of their usage.
3.2 Blocking Send and Receive Operations

A communicator specifies the communication context for a communication operation. Each communication context provides a separate “communication universe”: messages are always received within the context they were sent, and messages sent in different contexts do not interfere.

The communicator also specifies the set of processes that share this communication context. This process group is ordered and processes are identified by their rank within this group. Thus, the range of valid values for dest is 0, …, n-1, where n is the number of processes in the group. (If the communicator is an inter-communicator, then destinations are identified by their rank in the remote group. See Chapter 6.)

A predefined communicator MPI_COMM_WORLD is provided by MPI. It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI_COMM_WORLD.

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable MPI_COMM_WORLD as the comm argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 6. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (End of advice to users.)

Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (End of advice to implementors.)

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

```c
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

- **OUT** buf: initial address of receive buffer (choice)
- **IN** count: number of elements in receive buffer (non-negative integer)
- **IN** datatype: datatype of each receive buffer element (handle)
- **IN** source: rank of source (integer)
- **IN** tag: message tag (integer)
- **IN** comm: communicator (handle)
- **OUT** status: status object (Status)

```c
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)
```

Advice to users. Users that are comfortable with the notion of a flat name space for processes, and a single communication context, as offered by most existing communication libraries, need only use the predefined variable MPI_COMM_WORLD as the comm argument. This will allow communication with all the processes available at initialization time.

Users may define new communicators, as explained in Chapter 6. Communicators provide an important encapsulation mechanism for libraries and modules. They allow modules to have their own disjoint communication universe and their own process numbering scheme. (End of advice to users.)

Advice to implementors. The message envelope would normally be encoded by a fixed-length message header. However, the actual encoding is implementation dependent. Some of the information (e.g., source or destination) may be implicit, and need not be explicitly carried by messages. Also, processes may be identified by relative ranks, or absolute ids, etc. (End of advice to implementors.)

3.2.4 Blocking Receive

The syntax of the blocking receive operation is given below.

```c
MPI_RECV (buf, count, datatype, source, tag, comm, status)
```

- **OUT** buf: initial address of receive buffer (choice)
- **IN** count: number of elements in receive buffer (non-negative integer)
- **IN** datatype: datatype of each receive buffer element (handle)
- **IN** source: rank of source (integer)
- **IN** tag: message tag (integer)
- **IN** comm: communicator (handle)
- **OUT** status: status object (Status)

```c
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)
```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)

void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const

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

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

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

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

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

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

The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the source, tag and comm values specified by the receive operation. The receiver may specify a wildcard MPI_ANY_SOURCE value for source, and/or a wildcard MPI_ANY_TAG value for tag, indicating that any source and/or tag are acceptable. It cannot specify a wildcard value for comm. Thus, a message can be received by a receive operation only if it is addressed to the receiving process, has a matching communicator, has matching source unless source=MPI_ANY_SOURCE in the pattern, and has a matching tag unless tag=MPI_ANY_TAG in the pattern.

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

Note the asymmetry between send and receive operations: A receive operation may accept messages from an arbitrary sender, on the other hand, a send operation must specify
3.2. BLOCKING SEND AND RECEIVE OPERATIONS

a unique receiver. This matches a “push” communication mechanism, where data transfer is effected by the sender (rather than a “pull” mechanism, where data transfer is effected by the receiver).

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

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

3.2.5 Return Status

The source or tag of a received message may not be known if wildcard values were used in the receive operation. Also, if multiple requests are completed by a single MPI function (see Section 3.7.5), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_RECV. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects.

In C, status is a structure that contains three fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR: the structure may contain additional fields. Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

In Fortran, status is an array of INTEGERs of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

In C++, the status object is handled through the following methods:

```cpp
int MPI::Status::Get_source() const
void MPI::Status::Set_source(int source)
int MPI::Status::Get_tag() const
void MPI::Status::Set_tag(int tag)
int MPI::Status::Get_error() const
void MPI::Status::Set_error(int error)
```

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

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

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to `MPI_GET_COUNT` is required to "decode" this information.

```
MPI_GET_COUNT(status, datatype, count)
```

| IN  | status          | return status of receive operation (Status) |
| IN  | datatype        | datatype of each receive buffer entry (handle) |
| OUT | count           | number of received entries (integer) |

`int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`

Rationale. Some message-passing libraries use `INOUT count`, `tag`, and `source` arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with `INOUT` argument (e.g., using the `MPI_ANY_TAG` constant as the tag in a receive). Some libraries use calls that refer implicitly to the "last message received." This is not thread safe.

The `datatype` argument is passed to `MPI_GET_COUNT` so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to `MPI_PROBE` or `MPI_IPROBE`. With a status from `MPI_PROBE` or `MPI_IPROBE`, the same datatypes are allowed as in a call to `MPI_RECV` to receive this message. *(End of rationale.)*

The value returned as the `count` argument of `MPI_GET_COUNT` for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, `MPI_UNDEFINED` is returned.

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

The status argument also returns information on the length of the message received. However, this information is not directly available as a field of the status variable and a call to `MPI_GET_COUNT` is required to "decode" this information.

```
MPI_GET_COUNT(status, datatype, count)
```

| IN  | status          | return status of receive operation (Status) |
| IN  | datatype        | datatype of each receive buffer entry (handle) |
| OUT | count           | number of received entries (integer) |

`int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`

Rationale. Some message-passing libraries use `INOUT count`, `tag`, and `source` arguments, thus using them both to specify the selection criteria for incoming messages and return the actual envelope values of the received message. The use of a separate status argument prevents errors that are often attached with `INOUT` argument (e.g., using the `MPI_ANY_TAG` constant as the tag in a receive). Some libraries use calls that refer implicitly to the "last message received." This is not thread safe.

The `datatype` argument is passed to `MPI_GET_COUNT` so as to improve performance. A message might be received without counting the number of elements it contains, and the count value is often not needed. Also, this allows the same function to be used after a call to `MPI_PROBE` or `MPI_IPROBE`. With a status from `MPI_PROBE` or `MPI_IPROBE`, the same datatypes are allowed as in a call to `MPI_RECV` to receive this message. *(End of rationale.)*

The value returned as the `count` argument of `MPI_GET_COUNT` for a datatype of length zero where zero bytes have been transferred is zero. If the number of bytes transferred is greater than zero, `MPI_UNDEFINED` is returned.

Rationale. Zero-length datatypes may be created in a number of cases. An important case is `MPI_TYPE_CREATE_DARRAY`, where the definition of the particular darray results in an empty block on some MPI process. Programs written in an SPMD style will not check for this special case and may want to use `MPI_GET_COUNT` to check the status. *(End of rationale.*)
Advice to users. The buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. In most cases, the safest approach is to use the same datatype with MPI\_GET\_COUNT and the receive. (End of advice to users.)

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

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

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

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

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

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

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

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

There are no C++ bindings for MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE. To allow an OUT or INOUT MPI::Status argument to be ignored, all MPI C++ bindings that have OUT or INOUT MPI::Status parameters are overloaded with a second version that omits the OUT or INOUT MPI::Status parameter.

**Example 3.1** The C++ bindings for MPI\_PROBE are:

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

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

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

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

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

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

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

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

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

There are no C++ bindings for MPI\_STATUS\_IGNORE or MPI\_STATUSES\_IGNORE. To allow an OUT or INOUT MPI::Status argument to be ignored, all MPI C++ bindings that have OUT or INOUT MPI::Status parameters are overloaded with a second version that omits the OUT or INOUT MPI::Status parameter.

**Example 3.1** The C++ bindings for MPI\_PROBE are:
3.3 Data Type Matching and Data Conversion

3.3.1 Type Matching Rules

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

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

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

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

The types of a send and receive match (phase two) if both operations use identical names. That is, MPI\_INTEGER matches MPI\_INTEGER, MPI\_REAL matches MPI\_REAL, and so on. There is one exception to this rule, discussed in Section 4.2, the type MPI\_PACKED can match any other type.

The type of a variable in a host program matches the type specified in the communication operation if the datatype name used by that operation corresponds to the basic type of the host program variable. For example, an entry with type name MPI\_INTEGER matches a Fortran variable of type INTEGER. A table giving this correspondence for Fortran and C appears in Section 3.2.2. There are two exceptions to this last rule: an entry with type name MPI\_BYTE or MPI\_PACKED can be used to match any byte of storage (on a byte-addressable machine), irrespective of the datatype of the variable that contains this byte. The type MPI\_PACKED is used to send data that has been explicitly packed, or receive data that will be explicitly unpacked, see Section 4.2. The type MPI\_BYTE allows one to transfer the binary value of a byte in memory unchanged.

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

- Communication of typed values (e.g., with datatype different from MPI\_BYTE), where the datatypes of the corresponding entries in the sender program, in the send call, in the receive call and in the receiver program must all match.
- Communication of untyped values (e.g., of datatype MPI\_BYTE), where both sender and receiver use the datatype MPI\_BYTE. In this case, there are no requirements on the types of the corresponding entries in the sender and the receiver programs, nor is it required that they be the same.
- Communication involving packed data, where MPI\_PACKED is used.
3.3. DATA TYPE MATCHING AND DATA CONVERSION

The following examples illustrate the first two cases.

**Example 3.2** Sender and receiver specify matching types.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

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

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

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

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

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

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

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

**Advice to users.** If a buffer of type MPI_BYTE is passed as an argument to MPI_SEND, then MPI will send the data stored at contiguous locations, starting from the address indicated by the buf argument. This may have unexpected results when the data layout is not as a casual user would expect it to be. For example, some Fortran compilers implement variables of type CHARACTER as a structure that contains the character length and a pointer to the actual string. In such an environment, sending and receiving a Fortran CHARACTER variable using the MPI_BYTE type will not have the anticipated result of transferring the character string. For this reason, the user is advised to use typed communications whenever possible. (*End of advice to users.*)
CHAPTER 3. POINT-TO-POINT COMMUNICATION

3.3.2 Data Conversion

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

type conversion changes the datatype of a value, e.g., by rounding a REAL to an INTEGER.
representation conversion changes the binary representation of a value, e.g., from Hex floating point to IEEE floating point.

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical or character values, and to convert a floating point value to the nearest value that can be represented on the target system. Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

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

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment. Consider the three examples, 3.2-3.4. The first program is correct, assuming that $a$ and $b$ are REAL arrays of size $\geq 10$. If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

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

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

Advice to implementers. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion. Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (End of advice to implementers.)

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

The type matching rules imply that MPI communication never entails type conversion. On the other hand, MPI requires that a representation conversion be performed when a typed value is transferred across environments that use different representations for the datatype of this value. MPI does not specify rules for representation conversion. Such conversion is expected to preserve integer, logical or character values, and to convert a floating point value to the nearest value that can be represented on the target system. Overflow and underflow exceptions may occur during floating point conversions. Conversion of integers or characters may also lead to exceptions when a value that can be represented in one system cannot be represented in the other system. An exception occurring during representation conversion results in a failure of the communication. An error occurs either in the send operation, or the receive operation, or both.

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

No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment. Consider the three examples, 3.2-3.4. The first program is correct, assuming that $a$ and $b$ are REAL arrays of size $\geq 10$. If the sender and receiver execute in different environments, then the ten real values that are fetched from the send buffer will be converted to the representation for reals on the receiver site before they are stored in the receive buffer. While the number of real elements fetched from the send buffer equal the number of real elements stored in the receive buffer, the number of bytes stored need not equal the number of bytes loaded. For example, the sender may use a four byte representation and the receiver an eight byte representation for reals.

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

Data representation conversion also applies to the envelope of a message: source, destination and tag are all integers that may need to be converted.

Advice to implementers. The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion. Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (End of advice to implementers.)
MPI requires support for inter-language communication, i.e., if messages are sent by a C or C++ process and received by a Fortran process, or vice-versa. The behavior is defined in Section 16.3 on page 478.

### 3.4 Communication Modes

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

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

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

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

#### Rationale.

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

There are three additional communication modes.

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

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

There are three additional communication modes.

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

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

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

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

MPI_BSEND (buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (non-negative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<intype> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

void MPI::Bsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Send in buffered mode.

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

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

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

MPI_BSEND (buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (non-negative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<intype> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

void MPI::Bsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Send in buffered mode.
MPI_SSEND (buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (non-negative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

int MPI_Ssend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MPI_SSEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*), COUNT, DATATYPE, DEST, TAG, COMM, IERROR
void MPI::Comm::Ssend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Send in synchronous mode.

MPI_RSEND (buf, count, datatype, dest, tag, comm)

IN buf initial address of send buffer (choice)
IN count number of elements in send buffer (non-negative integer)
IN datatype datatype of each send buffer element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)

int MPI_Rsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MPI_RSEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*), COUNT, DATATYPE, DEST, TAG, COMM, IERROR
void MPI::Comm::Rsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Send in ready mode.

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

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.
Order Messages are non-overtaking: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the calls described later, such as MPI_CANCELL or MPI_WAITANY, are additional sources of nondeterminism.)

If a process has a single thread of execution, then any two communications executed by this process are ordered. On the other hand, if the process is multi-threaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are logically concurrent, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are logically concurrent receive two successively sent messages, then the two messages can match the two receives in either order.

Example 3.6 An example of non-overtaking messages.

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

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

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

Example 3.7 An example of two, intertwined matching pairs.

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

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

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

Example 3.7 An example of two, intertwined matching pairs.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```
Both processes invoke their first communication call. Since the first send of process zero uses the buffered mode, it must complete, irrespective of the state of process one. Since no matching receive is posted, the message will be copied into buffer space. (If insufficient buffer space is available, then the program will fail.) The second send is then invoked. At that point, a matching pair of send and receive operation is enabled, and both operations must complete. Process one next invokes its second receive call, which will be satisfied by the buffered message. Note that process one received the messages in the reverse order they were sent.

**Example 3.8** An exchange of messages.

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

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

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

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signalled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow will result. Additional synchronization has to be added to the program so as to prevent this from occurring. If standard sends are used, then the producer will be automatically throttled, as its send operations will block when buffer space is unavailable.

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

**Example 3.8** An exchange of messages.

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

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

Example 3.9 An errant attempt to exchange messages.

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

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

Example 3.10 An exchange that relies on buffering.

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

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

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

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

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

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

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

3.6 Buffer Allocation and Usage

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

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

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

```
MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
<type> BUFFER(*) INTEGER SIZE, IERROR
```

```
void MPI::Attach_buffer(void* buffer, int size)
```

Provides to MPI a buffer in the user’s memory to be used for buffering outgoing messages. The buffer is used only by messages sent in buffered mode. Only one buffer can be attached to a process at a time.

```
MPI_BUFFER_DETACH(buffer_addr, size)
OUT buffer_addr initial buffer address (choice)
OUT size buffer size, in bytes (non-negative integer)
```

```
int MPI_Buffer_detach(void* buffer_addr, int* size)
```

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

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

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

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

Example 3.11 Calls to attach and detach buffers.

#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach(&buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */

Advice to users. Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value.

(End of advice to users.)

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

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

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

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

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

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)

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

Example 3.11 Calls to attach and detach buffers.

#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(BUFFSIZE), BUFFSIZE);
/* a buffer of 10000 bytes can now be used by MPI_Bsend */
MPI_Buffer_detach(&buff, &size);
/* Buffer size reduced to zero */
MPI_Buffer_attach( buff, size);
/* Buffer of 10000 bytes available again */

Advice to users. Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value.

(End of advice to users.)

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

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

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

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

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

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use nonblocking communication. A nonblocking send start call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate send complete call is needed
dedicated to one sender-receiver pair, or be shared by all communications; buffering can be done in real or in virtual memory; it can use dedicated memory, or memory shared by other processes; buffer space may be allocated statically or be changed dynamically; etc. It does not seem feasible to provide a portable mechanism for querying or controlling buffering that would be compatible with all these choices, yet provide meaningful information. (End of rationale.)

3.6.1 Model Implementation of Buffered Mode

The model implementation uses the packing and unpacking functions described in Section 4.2 and the nonblocking communication functions described in Section 3.7.

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

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

- Traverse sequentially the PME queue from head towards the tail, deleting all entries for communications that have completed, up to the first entry with an uncompleted request; update queue head to point to that entry.
- Compute the number, \( n \), of bytes needed to store an entry for the new message. An upper bound on \( n \) can be computed as follows: A call to the function \( \text{MPIPACK}_\text{SIZE}(\text{count, datatype, comm, size}) \), with the \( \text{count}, \text{datatype} \) and \( \text{comm} \) arguments used in the \( \text{MPI_BSEND} \) call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant \( \text{MPI_BSEND}_\text{OVERHEAD} \) provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of \( n \) bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; \( \text{MPI_PACK} \) is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

3.7 Nonblocking Communication

One can improve performance on many systems by overlapping communication and computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller. Light-weight threads are one mechanism for achieving such overlap. An alternative mechanism that often leads to better performance is to use nonblocking communication. A nonblocking send start call initiates the send operation, but does not complete it. The send start call can return before the message was copied out of the send buffer. A separate send complete call is needed...
to complete the communication, i.e., to verify that the data has been copied out of the send buffer. With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done at the sender after the send was initiated and before it completed. Similarly, a nonblocking receive start call initiates the receive operation, but does not complete it. The call can return before a message is stored into the receive buffer. A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer. With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed. The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

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

The send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode. If the send mode is synchronous, then the send can complete only if a matching receive has started. That is, a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, nonblocking send may complete, if matched by a nonblocking receive, before the receive complete call occurs. (It can complete as soon as the sender “knows” the transfer will complete, but before the receiver “knows” the transfer will complete.) If the send mode is buffered then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive.

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

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

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

Nonblocking sends in the buffered and ready modes have a more limited impact. A nonblocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of nonblocking sends is advantageous in these cases only if data copying can be concurrent with computation.
3.7. NONBLOCKING COMMUNICATION

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

3.7.1 Communication Request Objects

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

3.7.2 Communication Initiation

We use the same naming conventions as for blocking communication: a prefix of B, S, or R is used for buffered, synchronous or ready mode. In addition a prefix of I (for immediate) indicates that the call is nonblocking.

MPI::ISEND(buf, count, datatype, dest, tag, comm, request)

Start a standard mode, nonblocking send.
**CHAPTER 3. POINT-TO-POINT COMMUNICATION**

MPI\_IBSEND(buf, count, datatype, dest, tag, comm, request)

1. \textbf{IN} buf \hspace{1cm} initial address of send buffer (choice)
2. \textbf{IN} count \hspace{1cm} number of elements in send buffer (non-negative integer)
3. \textbf{IN} datatype \hspace{1cm} datatype of each send buffer element (handle)
4. \textbf{IN} dest \hspace{1cm} rank of destination (integer)
5. \textbf{IN} tag \hspace{1cm} message tag (integer)
6. \textbf{IN} comm \hspace{1cm} communicator (handle)
7. \textbf{OUT} request \hspace{1cm} communication request (handle)

Start a buffered mode, nonblocking send.

```
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

MPI::Request MPI::Comm::Ibsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Start a buffered mode, nonblocking send.

**CHAPTER 3. POINT-TO-POINT COMMUNICATION**

MPI\_ISSSEND(buf, count, datatype, dest, tag, comm, request)

1. \textbf{IN} buf \hspace{1cm} initial address of send buffer (choice)
2. \textbf{IN} count \hspace{1cm} number of elements in send buffer (non-negative integer)
3. \textbf{IN} datatype \hspace{1cm} datatype of each send buffer element (handle)
4. \textbf{IN} dest \hspace{1cm} rank of destination (integer)
5. \textbf{IN} tag \hspace{1cm} message tag (integer)
6. \textbf{IN} comm \hspace{1cm} communicator (handle)
7. \textbf{OUT} request \hspace{1cm} communication request (handle)

Start a synchronous mode, nonblocking send.

```
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

MPI::Request MPI::Comm::Issend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Start a synchronous mode, nonblocking send.
3.7. NONBLOCKING COMMUNICATION

MPI_IRSEND(buf, count, datatype, dest, tag, comm, request)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>buf</td>
<td>initial address of send buffer (choice)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements in send buffer (non-negative integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each send buffer element (handle)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>dest</td>
<td>rank of destination (integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag (integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
<td>communication request (handle)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int MPI_Irsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

MPI::Request MPI::Comm::Irsend(const void* buf, int count, const MPI::Datatype& datatype, int dest, int tag) const

Start a ready mode nonblocking send.

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>buf</td>
<td>initial address of receive buffer (choice)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements in receive buffer (non-negative integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>datatype of each receive buffer element (handle)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>source</td>
<td>rank of source (integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag (integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
<td>communication request (handle)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
```

MPI::Request MPI::Comm::Irecv(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const

Start a nonblocking receive.
These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

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

3.7.3 Communication Completion

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

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

We shall use the following terminology: A null handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are inactive if the request is not associated with any ongoing communication (see Section 3.9). A handle is active if it is neither null nor inactive. An empty status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and count = 0 and MPI_TEST_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions (MPI_TEST_WAIT)\{ALL\SOME\ANY\}, where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS, and the returned status can be queried by the call MPI_TEST_CANCELLED.

Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only by the MPI completion functions that take arrays of MPI_STATUS. For the functions MPI_TEST, MPI_TESTANY, MPI_WAIT, and MPI_WAITANY, which return a single MPI_STATUS value, the normal MPI error return process should be used (not the MPI_ERROR field in the MPI_STATUS argument).

These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion.

A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation is called, until the send completes.

A nonblocking receive call indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

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

3.7.3 Communication Completion

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

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

We shall use the following terminology: A null handle is a handle with value MPI_REQUEST_NULL. A persistent request and the handle to it are inactive if the request is not associated with any ongoing communication (see Section 3.9). A handle is active if it is neither null nor inactive. An empty status is a status which is set to return tag = MPI_ANY_TAG, source = MPI_ANY_SOURCE, error = MPI_SUCCESS, and count = 0 and MPI_TEST_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

The fields in a status object returned by a call to MPI_WAIT, MPI_TEST, or any of the other derived functions (MPI_TEST_WAIT)\{ALL\SOME\ANY\}, where the request corresponds to a send call, are undefined, with two exceptions: The error status field will contain valid information if the wait or test call returned with MPI_ERR_IN_STATUS, and the returned status can be queried by the call MPI_TEST_CANCELLED.

Error codes belonging to the error class MPI_ERR_IN_STATUS should be returned only by the MPI completion functions that take arrays of MPI_STATUS. For the functions MPI_TEST, MPI_TESTANY, MPI_WAIT, and MPI_WAITANY, which return a single MPI_STATUS value, the normal MPI error return process should be used (not the MPI_ERROR field in the MPI_STATUS argument).
### 3.7. Nonblocking Communication

**MPI_Wait(request, status)**

- **INOUT** request (handle)
- **OUT** status object (Status)

```plaintext
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

**MPI_Wait(REQUEST, STATUS, IERROR)**

- **INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR**

```plaintext
void MPI::Request::Wait(MPI::Status& status)
```

A call to MPI_Wait returns when the operation identified by request is complete. If the communication object associated with this request was created by a nonblocking send or receive call, then the object is deallocated by the call to MPI_Wait and the request handle is set to MPI_REQUEST_NULL. MPI_Wait is a non-local operation.

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

One is allowed to call MPI_Wait with a null or inactive request argument. In this case the operation returns immediately with empty status.

**Advice to users.** Successful return of MPI_Wait after a MPI_IBSEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL (always being able to free program space that was committed to the communication subsystem). (End of advice to users.)

**Advice to implementors.** In a multi-threaded environment, a call to MPI_Wait should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (End of advice to implementors.)

**MPI_Test(request, flag, status)**

- **INOUT** request (handle)
- **OUT** flag (true if operation completed (logical))
- **OUT** status object (Status)

```plaintext
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

**MPI_TEST(REQUEST, FLAG, STATUS, IERROR)**

- **LOGICAL REQUEST, STATUS(MPI_STATUS_SIZE), IERROR**

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

One is allowed to call MPI_Test with a null or inactive request argument. In this case the operation returns immediately with empty status.

**Advice to users.** Successful return of MPI_Test after a MPI_IBSEND implies that the user send buffer can be reused — i.e., data has been sent out or copied into a buffer attached with MPI_BUFFER_ATTACH. Note that, at this point, we can no longer cancel the send (see Section 3.8). If a matching receive is never posted, then the buffer cannot be freed. This runs somewhat counter to the stated goal of MPI_CANCEL (always being able to free program space that was committed to the communication subsystem). (End of advice to users.)

**Advice to implementors.** In a multi-threaded environment, a call to MPI_Test should block only the calling thread, allowing the thread scheduler to schedule another thread for execution. (End of advice to implementors.)
bool MPI::Request::Test(MPI::Status& status)
bool MPI::Request::Test()

A call to MPI_TEST returns flag = true if the operation identified by request is complete. In such a case, the status object is set to contain information on the completed operation; if the communication object was created by a nonblocking send or receive, then it is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag = false, otherwise. In this case, the value of the status object is undefined. MPI_TEST is a local operation.

The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5. The status object for a send operation carries information that can be accessed by a call to MPI_TEST_CANCELLED (see Section 3.8).

One is allowed to call MPI_TEST with a null or inactive request argument. In such a case the operation returns with flag = true and empty status.

The functions MPI_WAIT and MPI_TEST can be used to complete both sends and receives.

Advice to users. The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to users.)

Rationale. The function MPI_TEST returns with flag = true exactly in those situations where the function MPI_WAIT returns; both functions return in such case the same value in status. Thus, a blocking Wait can be easily replaced by a nonblocking Test. (End of rationale.)

Example 3.12 Simple usage of nonblocking operations and MPI_WAIT.

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
**** do some computation to mask latency ****
CALL MPI_WAIT(request, status, ierr)
ELSE IF (rank.EQ.1) THEN
CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
**** do some computation to mask latency ****
CALL MPI_WAIT(request, status, ierr)
END IF

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

MPI_REQUEST_FREE(request)

Advice to users. The use of the nonblocking MPI_TEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_TEST. (End of advice to users.)

Rationale. The function MPI_TEST returns with flag = true exactly in those situations where the function MPI_WAIT returns; both functions return in such case the same value in status. Thus, a blocking Wait can be easily replaced by a nonblocking Test. (End of rationale.)

Example 3.12 Simple usage of nonblocking operations and MPI_WAIT.

CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
**** do some computation to mask latency ****
CALL MPI_WAIT(request, status, ierr)
ELSE IF (rank.EQ.1) THEN
CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
**** do some computation to mask latency ****
CALL MPI_WAIT(request, status, ierr)
END IF

A request object can be deallocated without waiting for the associated communication to complete, by using the following operation.

MPI_REQUEST_FREE(request)
3.7. NONBLOCKING COMMUNICATION

Example 3.13 An example using MPI_REQUEST_FREE.

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
IF (rank.EQ.0) THEN
  DO i=1, n-1
    CALL MPI_ISEND(outval, 1, MPI_REAL, 0, 0, MPI_COMM_WORLD, req, ierr)
    CALL MPI_WAIT(req, status, ierr)
  END DO
ELSE IF (rank.EQ.1) THEN
  CALL MPI_ISEND(outval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
  CALL MPI_IRECV(inval, 1, MPI_REAL, 1, 0, MPI_COMM_WORLD, req, ierr)
  CALL MPI_WAIT(req, status, ierr)
END IF

Mark the request object for deallocation and set request to MPI_REQUEST_NULL. An ongoing communication that is associated with the request will be allowed to complete. The request will be deallocated only after its completion.

Rationale. The MPI_REQUEST_FREE mechanism is provided for reasons of performance and convenience on the sending side. (End of rationale.)

Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using MPI_REQUEST_FREE. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of MPI_REQUEST_FREE practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (End of advice to users.)

Example 3.13 An example using MPI_REQUEST_FREE.

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

Advice to users. Once a request is freed by a call to MPI_REQUEST_FREE, it is not possible to check for the successful completion of the associated communication with calls to MPI_WAIT or MPI_TEST. Also, if an error occurs subsequently during the communication, an error code cannot be returned to the user — such an error must be treated as fatal. Questions arise as to how one knows when the operations have completed when using MPI_REQUEST_FREE. Depending on the program logic, there may be other ways in which the program knows that certain operations have completed and this makes usage of MPI_REQUEST_FREE practical. For example, an active send request could be freed when the logic of the program is such that the receiver sends a reply to the message sent — the arrival of the reply informs the sender that the send has completed and the send buffer can be reused. An active receive request should never be freed as the receiver will have no way to verify that the receive has completed and the receive buffer can be reused. (End of advice to users.)
3.7.4 Semantics of Nonblocking Communications

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

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

Example 3.14 Message ordering for nonblocking operations.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
  CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
  CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_IRECV(a, 1, MPI_REAL, 0, MPI_ANY_TAG, comm, r1, ierr)
  CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1, status, ierr)
CALL MPI_WAIT(r2, status, ierr)
```

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

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

Example 3.15 An illustration of progress semantics.

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

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

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

3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI_WAITSOME or MPI_WAITEMPTY can be used to wait for the completion of one out of several operations. A call to MPI_WAITALL or MPI_WAITSTALL can be used to wait for all pending operations in a list. A call to MPI_WAITALL or MPI_WAITSTALL can be used to complete all enabled operations in a list.

MPI_WAITANY (count, array_of_requests, index, status)

IN count list length (non-negative integer)
INOUT array_of_requests array of requests (array of handles)
OUT index index of handle for operation that completed (integer)
OUT status status object (Status)

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)

INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR

static int MPI::Request::Waitany(int count, MPI::Request array_of_requests[], MPI::Status& status)

static int MPI::Request::Waitany(int count, MPI::Request array_of_requests[])

Blocks until one of the operations associated with the active requests in the array has completed. If more than one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing communication. (The array is indexed from zero in C, and from one in Fortran.) If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to MPI_REQUEST_NULL.

The array_of_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI_UNDEFINED, and a empty status.

The execution of MPI_WAITANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_WAIT(&array_of_requests[i], status), where i is the value returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAITANY with an array containing one active entry is equivalent to MPI_WAIT.

3.7. NONBLOCKING COMMUNICATION

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

3.7.5 Multiple Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list, rather than having to wait for a specific message. A call to MPI_WAITSOME or MPI_WAITEMPTY can be used to wait for the completion of one out of several operations. A call to MPI_WAITALL or MPI_WAITSTALL can be used to wait for all pending operations in a list. A call to MPI_WAITALL or MPI_WAITSTALL can be used to complete all enabled operations in a list.

MPI_WAITANY (count, array_of_requests, index, status)

IN count list length (non-negative integer)
INOUT array_of_requests array of requests (array of handles)
OUT index index of handle for operation that completed (integer)
OUT status status object (Status)

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status)

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)

INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, STATUS(MPI_STATUS_SIZE), IERROR

static int MPI::Request::Waitany(int count, MPI::Request array_of_requests[], MPI::Status& status)

static int MPI::Request::Waitany(int count, MPI::Request array_of_requests[])

Blocks until one of the operations associated with the active requests in the array has completed. If more then one operation is enabled and can terminate, one is arbitrarily chosen. Returns in index the index of that request in the array and returns in status the status of the completing communication. (The array is indexed from zero in C, and from one in Fortran.) If the request was allocated by a nonblocking communication operation, then it is deallocated and the request handle is set to MPI_REQUEST_NULL.

The array_of_requests list may contain null or inactive handles. If the list contains no active handles (list has length zero or all entries are null or inactive), then the call returns immediately with index = MPI_UNDEFINED, and a empty status.

The execution of MPI_WAITANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_WAIT(&array_of_requests[i], status), where i is the value returned by index (unless the value of index is MPI_UNDEFINED). MPI_WAITANY with an array containing one active entry is equivalent to MPI_WAIT.
MPI_TESTANY(count, array_of_requests, index, flag, status)

Tests for completion of either one or none of the operations associated with active handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and returns in status the status of that operation; if the request was allocated by a nonblocking communication call then the request is deallocated and the handle is set to MPI_REQUEST_NULL. (The array is indexed from zero in C, and from one in Fortran.)

In the latter case (no operation completed), it returns flag = false, returns a value of MPI_UNDEFINED in index and status is undefined.

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

If the array of requests contains active handles then the execution of MPI_TESTANY(count, array_of_requests, index, status) has the same effect as the execution of MPI_TEST(&array_of_requests[i], flag, status), for i=0, 1, ..., count-1, in some arbitrary order, until one call returns flag = true, or all fail. In the former case, index is set to the last value of i, and in the latter case, it is set to MPI_UNDEFINED. MPI_TESTANY with an array containing one active entry is equivalent to MPI_TEST.

Rationale. The function MPI_TESTANY returns with flag = true exactly in those situations where the function MPI_WAITANY returns; both functions return in that case the same values in the remaining parameters. Thus, a blocking MPI_WAITANY can be easily replaced by a nonblocking MPI_TESTANY. The same relation holds for the other pairs of Wait and Test functions defined in this section. (End of rationale.)
3.7. NONBLOCKING COMMUNICATION

<table>
<thead>
<tr>
<th>1</th>
<th>MPI_WAITALL( count, array_of_requests, array_of_statuses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>count</td>
</tr>
<tr>
<td>INOUT</td>
<td>array_of_requests</td>
</tr>
<tr>
<td>OUT</td>
<td>array_of_statuses</td>
</tr>
</tbody>
</table>

int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES, IERROR)

INTEGER COUNT, ARRAY_OF_REQUESTS(*), INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR

static void MPI::Request::Waitall(int count, MPI::Request array_of_requests[], MPI::Status array_of_statuses[])

static void MPI::Request::Waitall(int count, MPI::Request array_of_requests[])

Blocks until all communication operations associated with active handles in the list complete, and return the status of all these operations (this includes the case where no handle in the list is active). Both arrays have the same number of valid entries. The i-th entry in array_of_statuses is set to the return status of the i-th operation. Requests that were created by nonblocking communication operations are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. The list may contain null or inactive handles. The call sets to empty the status of each such entry.

The error-free execution of MPI_WAITALL(count, array_of_requests, array_of_statuses) has the same effect as the execution of

MPI_WAIT(&array_of_request[i], &array_of_statuses[i]), for i=0,..., count-1, in some arbitrary order. MPI_WAITALL with an array of length one is equivalent to MPI_WAIT.

When one or more of the communications completed by a call to MPI_WAITALL fail, it is desirable to return specific information on each communication. The function MPI_WAITALL will return in such case the error code MPI_ERR_IN_STATUS and will set the error field of each status to a specific error code. This code will be MPI_SUCCESS, if the specific communication completed; it will be another specific error code, if it failed; or it can be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_WAITALL will return MPI_SUCCESS if no request had an error, or will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

**Rationale.** This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (End of rationale.)
MPI_TESTALL(count, array_of_requests, flag, array_of_statuses)

IN  count           lists length (non-negative integer)
INOUT array_of_requests       array of requests (array of handles)
OUT   flag                 (logical)
OUT   array_of_statuses     array of status objects (array of Status)

Returns flag = true if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active handle request is set to the status of the corresponding communication; if the request was allocated by a nonblocking communication call then it is deallocated, and the handle is set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty.

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

Errors that occurred during the execution of MPI_TESTALL are handled as errors in MPI_WAITHALL.

MPI_WAITSOME(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN  incount          length of array_of_requests (non-negative integer)
INOUT array_of_requests       array of requests (array of handles)
OUT   outcount          number of completed requests (integer)
OUT   array_of_indices   array of indices of operations that completed (array of integers)
OUT   array_of_statuses  array of status objects for operations that completed (array of Status)

Returns flag = true if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active handle request is set to the status of the corresponding communication; if the request was allocated by a nonblocking communication call then it is deallocated, and the handle is set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty.

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

Errors that occurred during the execution of MPI_WAITSOME are handled as errors in MPI_WAITHALL.
3.7. NONBLOCKING COMMUNICATION

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

static int MPI::Request::Waitsome(int incount,
MPI::Request array_of_requests[], int array_of_indices[],
MPI::Status array_of_statuses[])

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests: the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to MPI_REQUEST_NULL.

If the list contains no active handles, then the call returns immediately with outcount = MPI_UNDEFINED.

When one or more of the communications completed by MPI_WAITSOME fails, then it is desirable to return specific information on each communication. The arguments outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI_ERR_IN_STATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES, ARRAY_OF_STATUSES)

IN incount length of array_of_requests (non-negative integer)
INOUT array_of_requests array of requests (array of handles)
OUT outcount number of completed requests (integer)
OUT array_of_indices array of indices of operations that completed (array of integers)
OUT array_of_statuses array of status objects for operations that completed (array of Statuses)

int MPI_Testsome(int incount, MPI_Request *array_of_requests,
int *outcount, int *array_of_indices,
MPI_Status *array_of_statuses)

3.7. NONBLOCKING COMMUNICATION

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

static int MPI::Request::Waitsome(int incount,
MPI::Request array_of_requests[], int array_of_indices[],
MPI::Status array_of_statuses[])

Waits until at least one of the operations associated with active handles in the list have completed. Returns in outcount the number of requests from the list array_of_requests that have completed. Returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests: the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations. If a request that completed was allocated by a nonblocking communication call, then it is deallocated, and the associated handle is set to MPI_REQUEST_NULL.

If the list contains no active handles, then the call returns immediately with outcount = MPI_UNDEFINED.

When one or more of the communications completed by MPI_WAITSOME fails, then it is desirable to return specific information on each communication. The arguments outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI_ERR_IN_STATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.
MPI_TESTSOME(INCOUNT, ARRAY_OF_REQUESTS, OUTCOUNT, ARRAY_OF_INDICES,
ARRAY_OF_STATUSES, IERROR)

static int MPI::Request::Testsome(int incount,
MPI::Request array_of_requests[], int array_of_indices[],
MPI::Status array_of_statuses[])

Behaves like MPI_WAITSOME, except that it returns immediately. If no operation has
completed it returns outcount = 0. If there is no active handle in the list it returns outcount
= MPI_UNDEFINED.

 MPI_TESTSOME is a local operation, which returns immediately, whereas
MPI_WAITSOME will block until a communication completes, if it was passed a list that
contains at least one active handle. Both calls fulfill a fairness requirement: If a request for
a receive repeatedly appears in a list of requests passed to MPI_WAITSOME or
MPI_TESTSOME, and a matching send has been posted, then the receive will eventually
succeed, unless the send is satisfied by another receive; and similarly for send requests.

Errors that occur during the execution of MPI_TESTSOME are handled as for
MPI_WAITSOME.

Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use
of MPI_TESTANY. The former returns information on all completed communications,
with the latter, a new call is required for each communication that completes.

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

Advice to implementors. MPI_TESTSOME should complete as many pending com-
communications as possible. (End of advice to implementors.)

Example 3.16 Client-server code (starvation can occur).

CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank .GT. 0) THEN  ! client code
  DO WHILE(.TRUE.)
    CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
    CALL MPI_WAIT(request, status, ierr)
  END DO
ELSE  ! rank=0 -- server code
  DO i=1, size-1
    CALL MPI_IRECV(a(1,i), n, MPI_REAL, i tag, 
      comm, request_list(i), ierr)
END IF
3.7. Nonblocking Communication

Example 3.17 Same code, using MPI_WAITSOME.

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

3.7.6 Non-destructive Test of status

This call is useful for accessing the information associated with a request, without freeing the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same completed request and extract from it the status information.
CHAPTER 3. POINT-TO-POINT COMMUNICATION

3.8 Probe and Cancel

The MPI_PROBE and MPI_IPROBE operations allow incoming messages to be checked for, without actually receiving them. The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status). In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

The MPICANCEL operation allows pending communications to be canceled. This is required for cleanup. Posting a send or a receive ties up user resources (send or receive buffers), and a cancel may be needed to free these resources gracefully.

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)

INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

LOGICAL FLAG

bool MPI::Request::Get_status(MPI::Status& status) const

bool MPI::Request::Get_status() const

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

bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const
bool MPI::Comm::Iprobe(int source, int tag) const

MPI_IPROBE(source, tag, comm, flag, status) returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the program, and returns in status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = false, and leaves status undefined.

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

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

The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag argument can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

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

MPI_PROBE(source, tag, comm, status)

IN source source rank, or MPI_ANY_SOURCE (integer)
IN tag tag value, or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (Status)

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
MPI_PROBE(SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR)
void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
void MPI::Comm::Probe(int source, int tag) const

MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns only after a matching message has been found.

The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress: if a call to MPI_PROBE has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to MPI_PROBE will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and

3.8. PROBE AND CANCEL

bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const
bool MPI::Comm::Iprobe(int source, int tag) const

MPI_IPROBE(source, tag, comm, flag, status) returns flag = true if there is a message that can be received and that matches the pattern specified by the arguments source, tag, and comm. The call matches the same message that would have been received by a call to MPI_RECV(..., source, tag, comm, status) executed at the same point in the program, and returns in status the same value that would have been returned by MPI_RECV(). Otherwise, the call returns flag = false, and leaves status undefined.

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

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

The source argument of MPI_PROBE can be MPI_ANY_SOURCE, and the tag argument can be MPI_ANY_TAG, so that one can probe for messages from an arbitrary source and/or with an arbitrary tag. However, a specific communication context must be provided with the comm argument.

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

MPI_PROBE(source, tag, comm, status)

IN source source rank, or MPI_ANY_SOURCE (integer)
IN tag tag value, or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (Status)

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
MPI_PROBE(SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR)
void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const
void MPI::Comm::Probe(int source, int tag) const

MPI_PROBE behaves like MPI_IPROBE except that it is a blocking call that returns only after a matching message has been found.

The MPI implementation of MPI_PROBE and MPI_IPROBE needs to guarantee progress: if a call to MPI_PROBE has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to MPI_PROBE will return, unless the message is received by another concurrent receive operation (that is executed by another thread at the probing process). Similarly, if a process busy waits with MPI_IPROBE and
a matching message has been issued, then the call to MPI_IPROBE will eventually return
flag = true unless the message is received by another concurrent receive operation.

Example 3.18 Use blocking probe to wait for an incoming message.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE IF (rank.EQ.2) THEN
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
    IF (status(MPI_SOURCE) .EQ. 0) THEN
      CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
    ELSE
      CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
    END IF
  END DO
END IF
END IF
```
Each message is received with the right type.

Example 3.19 A similar program to the previous example, but now it has a problem.

```fortran
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
  CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
ELSE IF (rank.EQ.1) THEN
  CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
ELSE IF (rank.EQ.2) THEN
  DO i=1, 2
    CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
    IF (status(MPI_SOURCE) .EQ. 0) THEN
      CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE, 0, comm, status, ierr)
    ELSE
      CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE, 0, comm, status, ierr)
    END IF
  END DO
END IF
END IF
```
We slightly modified example 3.18, using MPI_ANY_SOURCE as the source argument in
the two receive calls in statements labeled 100 and 200. The program is now incorrect: the
receive operation may receive a message that is distinct from the message probed by the
preceding call to MPI_PROBE.
3.8. PROBE AND CANCEL

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

MPI_CANCEL(request)

IN request communication request (handle)

int MPI_Cancel(MPI_Request *request)

MPI_CANCEL(REQUEST, IERROR)

INTEGER REQUEST, IERROR

void MPI::Request::Cancel() const

A call to MPI_CANCEL marks for cancellation a pending, nonblocking communication
operation (send or receive). The cancel call is local. It returns immediately, possibly before
the communication is actually canceled. It is still necessary to complete a communication
that has been marked for cancellation, using a call to MPI_REQUEST_FREE, MPI_WAIT or
MPI_TEST (or any of the derived operations).

If a communication is marked for cancellation, then a MPI_WAIT call for that
communication is guaranteed to return, irrespective of the activities of other processes (i.e.,
MPI_WAIT behaves as a local function); similarly if MPI_TEST is repeatedly called in a
busy wait loop for a canceled communication, then MPI_TEST will eventually be success-
ful.

MPI_CANCEL can be used to cancel a communication that uses a persistent request (see
Section 3.9), in the same way it is used for nonpersistent requests. A successful cancellation
cancels the active communication, but not the request itself. After the call to MPI_CANCEL
and the subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can
be activated for a new communication.

The successful cancellation of a buffered send frees the buffer space occupied by the
pending message.

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

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

MPI_TEST_CANCELLED(status, flag)

IN status status object (Status)

OUT flag (logical)

int MPI_Test_cancelled(MPI_Status *status, int *flag)

MPI_TEST_CANCELLED(status, flag, IERROR)

LOGICAL FLAG INTEGER STATUS(MPI_STATUS_SIZE), IERROR

bool MPI::Status::Is_cancelled() const

Returns flag = true if the communication associated with the status object was canceled successfully. In such a case, all other fields of status (such as count or tag) are undefined. Returns flag = false, otherwise. If a receive operation might be canceled then one should call MPI_TEST_CANCELLED first, to check whether the operation was canceled, before checking on the other fields of the return status.

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

Advice to implementors. If a send operation uses an “eager” protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement MPI_CANCEL, this is still a local operation, since its completion does not depend on the code executed by other processes. If processing is required on another process, this should be transparent to the application (hence the need for an interrupt and an interrupt handler). (End of advice to implementors.)

3.9 Persistent Communication Requests

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages. The persistent request thus created can be thought of as a communication port or a “half-channel.” It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another.
It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa.

A persistent communication request is created using one of the five following calls. These calls involve no communication.

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

<table>
<thead>
<tr>
<th>IN</th>
<th>buf</th>
<th>initial address of send buffer (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements sent (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>type of each element (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>dest</td>
<td>rank of destination (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
<td>communication request (handle)</td>
</tr>
</tbody>
</table>

```
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

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

<table>
<thead>
<tr>
<th>IN</th>
<th>buf</th>
<th>initial address of send buffer (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements sent (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>type of each element (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>dest</td>
<td>rank of destination (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>tag</td>
<td>message tag (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
<td>communication request (handle)</td>
</tr>
</tbody>
</table>

```
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
```
CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI::Prequest MPI::Comm::Bsend_init(const void* buf, int count, const
MPI::Datatype& datatype, int dest, int tag) const

Creates a persistent communication request for a buffered mode send.

MPI_SSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of elements sent (non-negative integer)
IN datatype type of each element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)

MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Prequest MPI::Comm::Ssend_init(const void* buf, int count, const
MPI::Datatype& datatype, int dest, int tag) const

Creates a persistent communication object for a synchronous mode send operation.

MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of elements sent (non-negative integer)
IN datatype type of each element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)

MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count, const
MPI::Datatype& datatype, int dest, int tag) const

Creates a persistent communication object for a synchronous mode send operation.

MPI_RSEND_INIT(buf, count, datatype, dest, tag, comm, request)

IN buf initial address of send buffer (choice)
IN count number of elements sent (non-negative integer)
IN datatype type of each element (handle)
IN dest rank of destination (integer)
IN tag message tag (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)

MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Prequest MPI::Comm::Rsend_init(const void* buf, int count, const
MPI::Datatype& datatype, int dest, int tag) const
3.9. PERSISTENT COMMUNICATION REQUESTS

Creates a persistent communication object for a ready mode send operation.

MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)

OUT buf initial address of receive buffer (choice)
IN count number of elements received (non-negative integer)
IN datatype type of each element (handle)
IN source rank of source or MPI_ANY_SOURCE (integer)
IN tag message tag or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT request communication request (handle)

int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
MPI::Request MPI::Comm::Recv_init(void* buf, int count, const MPI::Datatype& datatype, int source, int tag) const

Creates a persistent communication request for a receive operation. The argument buf
is marked as OUT because the user gives permission to write on the receive buffer by passing
the argument to MPI_RECV_INIT.

A persistent communication request is inactive after it was created — no active
communication is attached to the request.

A communication (send or receive) that uses a persistent request is initiated by the
function MPI_START.

MPI_START(request)

INOUT request communication request (handle)

int MPI_Start(MPI_Request *request)

MPI_START(REQUEST, IERROR)
INTEGER REQUEST, IERROR
void MPI::Prequest::Start()

The argument, request, is a handle returned by one of the previous five calls. The
associated request should be inactive. The request becomes active once the call is made.

If the request is for a send with ready mode, then a matching receive should be posted
before the call is made. The communication buffer should not be accessed after the call,
and until the operation completes.

The call is local, with similar semantics to the nonblocking communication operations
described in Section 3.7. That is, a call to MPI_START with a request created by

int MPI_Start(MPI_Request *request)

MPI_START(REQUEST, IERROR)
INTEGER REQUEST, IERROR
void MPI::Prequest::Start()

The argument, request, is a handle returned by one of the previous five calls. The
associated request should be inactive. The request becomes active once the call is made.

If the request is for a send with ready mode, then a matching receive should be posted
before the call is made. The communication buffer should not be accessed after the call,
and until the operation completes.

The call is local, with similar semantics to the nonblocking communication operations
described in Section 3.7. That is, a call to MPI_START with a request created by
MPI\_SEND\_INIT starts a communication in the same manner as a call to \texttt{MPI\_SEND}; a call to \texttt{MPI\_START} with a request created by \texttt{MPI\_BSEND\_INIT} starts a communication in the same manner as a call to \texttt{MPI\_IBSEND}; and so on.

\begin{verbatim}
CHAPTER 3. POINT-TO-POINT COMMUNICATION

MPI\_START\_ALL(count, array\_of\_requests)
IN    count    list length (non-negative integer)
INOUT  array\_of\_requests   array of requests (array of handle)

int MPI\_Start\_all(int count, MPI\_Request *array\_of\_requests)

MPI\_START\_ALL(COUNT, ARRAY\_OF\_REQUESTS, IERROR)
INTEGER COUNT, ARRAY\_OF\_REQUESTS(*), IERROR

static void MPI\::Pre\_request::Start\_all(int count,
MPI\::Pre\_request array\_of\_requests[])

Start all communications associated with requests in \texttt{array\_of\_requests}. A call to \texttt{MPI\_START\_ALL(count, array\_of\_requests)} has the same effect as calls to \texttt{MPI\_START(\{array\_of\_requests[i]\}]} for i=0,...,count-1, in some arbitrary order. A communication started with a call to \texttt{MPI\_START} or \texttt{MPI\_START\_ALL} is completed by a call to \texttt{MPI\_WAIT}, \texttt{MPI\_TEST}, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an \texttt{MPI\_START} or \texttt{MPI\_START\_ALL} call.

A persistent request is deallocated by a call to \texttt{MPI\_REQUEST\_FREE} (Section 3.7.3). The call to \texttt{MPI\_REQUEST\_FREE} can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes inactive. Active receive requests should not be freed. Otherwise, it will not be possible to check that the receive has completed. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form, ...
\end{verbatim}

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

A send operation initiated with \texttt{MPI\_START} can be matched with any receive operation and, likewise, a receive operation initiated with \texttt{MPI\_START} can receive messages generated by any send operation.

\begin{verbatim}
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 on pages 463 and 466. (End of advice to users.)
\end{verbatim}
3.10 Send-Receive

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues. The send-receive operation can be used in conjunction with the functions described in Chapter 7 in order to perform shifts on various logical topologies. Also, a send-receive operation is useful for implementing remote procedure calls.

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

MPI_SENDRECV(buf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN sendbuf initial address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative integer)
IN sendtype type of elements in send buffer (handle)
IN dest rank of destination (integer)
IN sendtag send tag (integer)
OUT recvbuf initial address of receive buffer (choice)
IN recvcount number of elements in receive buffer (non-negative integer)
IN recvtype type of elements in receive buffer (handle)
IN source rank of source (integer)
IN recvtag receive tag (integer)
IN comm communicator (handle)
OUT status status object (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)

MPI_SENDRECV(buf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror)

<intype> SENDBUF(*), RECVPBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVPRTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

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)

MPI_SENDRECV(buf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror)

<intype> SENDBUF(*), RECVPBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVPRTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf, int recvcount, const MPI::Datatype& recvtype, int source, int recvtag, MPI::Status& status) const

void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf, int recvcount, const MPI::Datatype& recvtype, int source, int recvtag) const

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.

MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

INOUT buf initial address of send and receive buffer (choice)
IN count number of elements in send and receive buffer (non-negative integer)
IN datatype type of elements in send and receive buffer (handle)
IN dest rank of destination (integer)
IN sendtag send message tag (integer)
IN source rank of source (integer)
IN recvtag receive message tag (integer)
IN comm communicator (handle)
OUT status status object (Status)

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)

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)

<int> BUF(*), COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Sendrecv_replace(void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag, MPI::Status& status) const

void MPI::Comm::Sendrecv_replace(void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag) const

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)

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)

<int> BUF(*), COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Sendrecv_replace(void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag, MPI::Status& status) const

void MPI::Comm::Sendrecv_replace(void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag) const
3.11. NULL PROCESSES

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

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

3.11 Null Processes

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

The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. A send to MPI_PROC_NULL succeeds and returns as soon as possible. A receive from MPI_PROC_NULL succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with source = MPI_PROC_NULL is executed then the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0.
Chapter 4

Datatypes

Basic datatypes were introduced in Section 3.2.2 Message Data on page 27 and in Section 3.3 Data Type Matching and Data Conversion on page 34. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

4.1 Derived Datatypes

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

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

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

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

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

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

Let

\[ T_{\text{typemap}} = \{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\} \]

be such a type map, where type\(_i\) are basic types, and disp\(_i\) are displacements. Let

\[ T_{\text{typesig}} = \{\text{type}_0, \ldots, \text{type}_{n-1}\} \]

be the associated type signature. This type map, together with a base address buf, specifies a communication buffer: the communication buffer that consists of \(n\) entries, where the \(i\)-th entry is at address \(\text{buf} + \text{disp}_i\) and has type \text{type}_i. A message assembled from such a communication buffer will consist of \(n\) values, of the types defined by \(T_{\text{typesig}}\).

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.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation \(\text{MPI\_SEND}(\text{buf}, 1, \text{datatype}, \ldots)\) will use the send buffer defined by the base address \text{buf} and the general datatype associated with \text{datatype}; it will generate a message with the type signature determined by the \(T_{\text{typesig}}\) argument. \(\text{MPI\_RECV}(\text{buf}, 1, \text{datatype}, \ldots)\) will use the receive buffer defined by the base address \text{buf} and the general datatype associated with \text{datatype}.

A message assembled from such a type map, where type\(_i\) are basic types, and disp\(_i\) are displacements. Let

\[ T_{\text{typemap}} = \{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\} \]

be the associated type signature. This type map, together with a base address buf, specifies a communication buffer: the communication buffer that consists of \(n\) entries, where the \(i\)-th entry is at address \(\text{buf} + \text{disp}_i\) and has type \text{type}_i. A message assembled from such a communication buffer will consist of \(n\) values, of the types defined by \(T_{\text{typesig}}\).

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.

We can use a handle to a general datatype as an argument in a send or receive operation, instead of a basic datatype argument. The operation \(\text{MPI\_SEND}(\text{buf}, 1, \text{datatype}, \ldots)\) will use the send buffer defined by the base address \text{buf} and the general datatype associated with \text{datatype}; it will generate a message with the type signature determined by the \(T_{\text{typesig}}\) argument. \(\text{MPI\_RECV}(\text{buf}, 1, \text{datatype}, \ldots)\) will use the receive buffer defined by the base address \text{buf} and the general datatype associated with \text{datatype}.

General datatypes can be used in all send and receive operations. We discuss, in Section 4.1.11, the case where the second argument count has value \(> 1\).

The basic datatypes presented in Section 3.2.2 are particular cases of a general datatype, and are predefined. Thus, \texttt{MPI\_INT} is a predefined handle to a datatype with type map \([(\text{int}, 0)]\), with one entry of type \text{int} and displacement zero. The other basic datatypes are similar.

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

\[ T_{\text{typemap}} = \{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\}, \]

then

\[
\begin{align*}
\text{lb}(T_{\text{typemap}}) &= \min_j \text{disp}_j, \\
\text{ub}(T_{\text{typemap}}) &= \max_j (\text{disp}_j + \text{sizeof}(\text{type}_j)) + \epsilon, \quad \text{and} \\
\text{extent}(T_{\text{typemap}}) &= \text{ub}(T_{\text{typemap}}) - \text{lb}(T_{\text{typemap}}).
\end{align*}
\]

(4.1)

If \text{type}_i\ requires alignment to a byte address that is a multiple of \(k_i\), then \(\epsilon\) is the least nonnegative increment needed to round \text{extent}(T_{\text{typemap}}) to the next multiple of \(\max k_i\). The complete definition of extent is given on page 96.
Example 4.1 Assume that $T_{\text{type}} = \{(\text{double}, 0), (\text{char}, 8)\}$ (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (End of rationale.)

4.1.2 Datatype Constructors

Contiguous The simplest datatype constructor is $\text{MPI\_TYPE\_CONTIGUOUS}$ which allows replication of a datatype into contiguous locations.

$\text{MPI\_TYPE\_CONTIGUOUS}(\text{count}, \text{oldtype}, \text{newtype})$

<table>
<thead>
<tr>
<th>IN</th>
<th>count</th>
<th>replication count (nonnegative integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>oldtype</td>
<td>old datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>newtype</td>
<td>new datatype (handle)</td>
</tr>
</tbody>
</table>

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

$\text{MPI\_TYPE\_CONTIGUOUS}($\text{COUNT}, $\text{OLDTYPE}, $\text{NEWTYPE}, $\text{ERROR}$)$

INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_contiguous(int count) const

e datatype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using extent as the size of the concatenated copies.

Example 4.2 Let oldtype have type map $\{(\text{double}, 0), (\text{char}, 8)\}$, with extent 16, and let count = 3. The type map of the datatype returned by newtype is $\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40)\}$; i.e., alternating double and char elements, with displacements 0, 8, 16, 24, 32, 40.

4.1. DERIVED DATATYPES

Example 4.1 Assume that $T_{\text{type}} = \{(\text{double}, 0), (\text{char}, 8)\}$ (a double at displacement zero, followed by a char at displacement eight). Assume, furthermore, that doubles have to be strictly aligned at addresses that are multiples of eight. Then, the extent of this datatype is 16 (9 rounded to the next multiple of 8). A datatype that consists of a character immediately followed by a double will also have an extent of 16.

Rationale. The definition of extent is motivated by the assumption that the amount of padding added at the end of each structure in an array of structures is the least needed to fulfill alignment constraints. More explicit control of the extent is provided in Section 4.1.6. Such explicit control is needed in cases where the assumption does not hold, for example, where union types are used. (End of rationale.)

4.1. Type Constructors with Explicit Addresses

In Fortran, the functions $\text{MPI\_TYPE\_CREATE\_HVVECTOR}$, $\text{MPI\_TYPE\_CREATE\_HINDEXED}$, $\text{MPI\_TYPE\_CREATE\_STRUCT}$, and $\text{MPI\_GET\_ADDRESS}$ 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 of type INTEGER*8.

4.1.2 Datatype Constructors

Contiguous The simplest datatype constructor is $\text{MPI\_TYPE\_CONTIGUOUS}$ which allows replication of a datatype into contiguous locations.

$\text{MPI\_TYPE\_CONTIGUOUS}($\text{count}, $\text{oldtype}, $\text{newtype}$)$

<table>
<thead>
<tr>
<th>IN</th>
<th>count</th>
<th>replication count (nonnegative integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>oldtype</td>
<td>old datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>newtype</td>
<td>new datatype (handle)</td>
</tr>
</tbody>
</table>

int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

$\text{MPI\_TYPE\_CONTIGUOUS}($\text{COUNT}, $\text{OLDTYPE}, $\text{NEWTYPE}, $\text{ERROR}$)$

INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_contiguous(int count) const

e datatype is the datatype obtained by concatenating count copies of oldtype. Concatenation is defined using extent as the size of the concatenated copies.

Example 4.2 Let oldtype have type map $\{(\text{double}, 0), (\text{char}, 8)\}$, with extent 16, and let count = 3. The type map of the datatype returned by newtype is $\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40)\}$; i.e., alternating double and char elements, with displacements 0, 8, 16, 24, 32, 40.
In general, assume that the type map of `oldtype` is

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

with extent `ex`. Then `newtype` has a type map with `count` entries defined by:

\[
\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1}), (type_0, disp_0 + ex), \ldots, (type_{n-1}, disp_{n-1} + ex),
\]

\[
\ldots, (type_0, disp_0 + ex \cdot (count - 1)), \ldots, (type_{n-1}, disp_{n-1} + ex \cdot (count - 1))\}.
\]

**Vector** The function `MPI_TYPE_VECTOR` is a more general constructor that allows replication of a datatype into locations that consist of equally spaced blocks. Each block is obtained by concatenating the same number of copies of the old datatype. The spacing between blocks is a multiple of the extent of the old datatype.

```c
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype) {
    int MPI_Type_vector(int count, int blocklength, int stride, IN oldtype, OUT newtype);
    MPI_Datatype oldtype, MPI_Datatype *newtype);
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR;
    MPI::Datatype MPI::Datatype::Create_vector(int count, int blocklength, int stride) const
```

**Example 4.3** Assume, again, that `oldtype` has type map \(\{(\text{double}, 0), (\text{char}, 8)\}\), with extent 16. A call to `MPI_TYPE_VECTOR(2, 3, 4, oldtype, newtype)` will create the datatype with type map:

\[
\{(\text{double}, 0), (\text{char}, 8), (\text{double}, 16), (\text{char}, 24), (\text{double}, 32), (\text{char}, 40),
\]

\[
(\text{double}, 64), (\text{char}, 72), (\text{double}, 80), (\text{char}, 88), (\text{double}, 96), (\text{char}, 104)\}.
\]

That is, two blocks with three copies each of the old type, with a stride of 4 elements (4·16 bytes) between the blocks.
Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,

\{(double, 0), (char, 8), (double, -32), (char, -24), (double, -64), (char, -56)\}.

In general, assume that oldtype has type map,

\{\langle type_0, disp_0\rangle, ..., \langle type_n, disp_n\rangle\},

with extent \(ex\). Let \(bl\) be the blocklength. The newly created datatype has a type map with count \(\cdot bl \cdot n\) entries:

\{\langle type_0, disp_0 + ex \rangle, ..., \langle type_n, disp_n + ex \rangle, ...

\langle type_0, disp_0 + (bl - 1) \cdot ex \rangle, ..., \langle type_n, disp_n + (bl - 1) \cdot ex \rangle,

\langle type_0, disp_0 + stride \cdot ex \rangle, ..., \langle type_n, disp_n + stride \cdot ex \rangle, ...

\langle type_0, disp_0 + (strid + bl - 1) \cdot ex \rangle, ..., \langle type_n, disp_n + (strid + bl - 1) \cdot ex \rangle, ...

\langle type_0, disp_0 + (\text{count} - 1) \cdot ex \rangle, ...,

\langle type_0, disp_0 + (strid \cdot (\text{count} - 1) \cdot ex) \rangle, ...,

\langle type_0, disp_0 + (\text{strid} \cdot (\text{count} - 1) + bl - 1) \cdot ex) \rangle, ...,

\langle type_0, disp_0 + (\text{strid} \cdot (\text{count} - 1) + bl - 1) \cdot ex) \rangle.

A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), \(n\) arbitrary.

HVECTOR The function MPI_TYPE_CREATE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for “heterogeneous”).

MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)

<table>
<thead>
<tr>
<th>IN</th>
<th>count</th>
<th>number of blocks (nonnegative integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>blocklength</td>
<td>number of elements in each block (nonnegative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>stride</td>
<td>number of bytes between start of each block (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>oldtype</td>
<td>old datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>newtype</td>
<td>new datatype (handle)</td>
</tr>
</tbody>
</table>

Example 4.4 A call to MPI_TYPE_VECTOR(3, 1, -2, oldtype, newtype) will create the datatype,

\{(double, 0), (char, 8), (double, -32), (char, -24), (double, -64), (char, -56)\}.

In general, assume that oldtype has type map,

\{\langle type_0, disp_0\rangle, ..., \langle type_n, disp_n\rangle\},

with extent \(ex\). Let \(bl\) be the blocklength. The newly created datatype has a type map with count \(\cdot bl \cdot n\) entries:

\{\langle type_0, disp_0\rangle, ..., \langle type_n, disp_n\rangle, ...

\langle type_0, disp_0 + ex \rangle, ..., \langle type_n, disp_n + ex \rangle, ...

\langle type_0, disp_0 + (bl - 1) \cdot ex \rangle, ..., \langle type_n, disp_n + (bl - 1) \cdot ex \rangle,

\langle type_0, disp_0 + stride \cdot ex \rangle, ..., \langle type_n, disp_n + stride \cdot ex \rangle, ...

\langle type_0, disp_0 + (strid + bl - 1) \cdot ex \rangle, ..., \langle type_n, disp_n + (strid + bl - 1) \cdot ex \rangle, ...

\langle type_0, disp_0 + (\text{count} - 1) \cdot ex \rangle, ...,

\langle type_0, disp_0 + (strid \cdot (\text{count} - 1) \cdot ex) \rangle, ...,

\langle type_0, disp_0 + (\text{strid} \cdot (\text{count} - 1) + bl - 1) \cdot ex) \rangle, ...,

\langle type_0, disp_0 + (\text{strid} \cdot (\text{count} - 1) + bl - 1) \cdot ex) \rangle.

A call to MPI_TYPE_CONTIGUOUS(count, oldtype, newtype) is equivalent to a call to MPI_TYPE_VECTOR(count, 1, 1, oldtype, newtype), or to a call to MPI_TYPE_VECTOR(1, count, n, oldtype, newtype), \(n\) arbitrary.

HVECTOR The function MPI_TYPE_CREATE_HVECTOR is identical to MPI_TYPE_VECTOR, except that stride is given in bytes, rather than in elements. The use for both types of vector constructors is illustrated in Section 4.1.14. (H stands for “heterogeneous”).

MPI_TYPE_CREATE_HVECTOR(count, blocklength, stride, oldtype, newtype)

<table>
<thead>
<tr>
<th>IN</th>
<th>count</th>
<th>number of blocks (nonnegative integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>blocklength</td>
<td>number of elements in each block (nonnegative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>stride</td>
<td>number of bytes between start of each block (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>oldtype</td>
<td>old datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>newtype</td>
<td>new datatype (handle)</td>
</tr>
</tbody>
</table>
The function `MPI::Datatype::Create_hvector(int count, int blocklength, MPI::Aint stride)` const

This function replaces `MPI_TYPE_HVECTOR`, whose use is deprecated. See also Chapter 15.

Assume that `oldtype` has type map,

```
{{type0, disp0}, ..., {type_{n-1}, disp_{n-1}}},
```

with extent `ex`. Let `bl` be the block length. The newly created datatype has a type map with `count · bl · n` entries:

```
{{type0, disp0}, ..., {type_{n-1}, disp_{n-1}}},
```

```
{type0, disp0 + ex}, ..., {type_{n-1}, disp_{n-1} + ex}, ...
```

```
{type0, disp0 + (bl - 1) · ex}, ..., {type_{n-1}, disp_{n-1} + (bl - 1) · ex},
```

```
{type0, disp0 + stride}, ..., {type_{n-1}, disp_{n-1} + stride}, ...
```

```
{type0, disp0 + stride + (bl - 1) · ex}, ...
```

```
{type_{n-1}, disp_{n-1} + stride + (bl - 1) · ex}, ...
```

```
{type0, disp0 + stride · (count - 1)}, ..., {type_{n-1}, disp_{n-1} + stride · (count - 1)}, ...
```

```
{type0, disp0 + stride · (count - 1) + (bl - 1) · ex}, ...
```

```
{type_{n-1}, disp_{n-1} + stride · (count - 1) + (bl - 1) · ex}.
```

Indexed The function `MPI_TYPE_INDEXED` allows replication of an old datatype into a sequence of blocks (each block is a concatenation of the old datatype), where each block can contain a different number of copies and have a different displacement. All block displacements are multiples of the old type extent.
In general, assume that oldtype has type map,

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

with extent \(ex\). Let \(B\) be the \textit{array_of_blocklengths} argument and \(D\) be the \textit{array_of_displacements} argument. The newly created datatype has \(n \cdot \sum_{i=0}^{\text{count} - 1} B[i]\) entries:

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

Example 4.5 Let oldtype have type map \{\{double, 0\}, \{char, 8\}\}, with extent 16. Let \(B = (3, 1)\) and let \(D = (4, 0)\). A call to \texttt{MPI_TYPE_INDEXED(2, B, D, oldtype, newtype)} returns a datatype with type map,

\[
\{(double, 64), \{char, 72\}, \{double, 80\}, \{char, 88\}, \{double, 96\}, \{char, 104\}, \\
\{double, 0\}, \{char, 8\}\}.
\]

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.
\[(\text{type}_0, \text{disp}_0 + (D[\text{count} - 1] + B[\text{count} - 1] - 1) \cdot ex), \ldots, \\
(\text{type}_{n-1}, \text{disp}_{n-1} + (D[\text{count} - 1] + B[\text{count} - 1] - 1) \cdot ex)\].

A call to \text{MPI\_TYPE\_VECTOR(count, blocklength, stride, oldtype, newtype)} is equivalent to a call to \text{MPI\_INDEXED(count, B, D, oldtype, newtype)} where
\[
D[j] = j \cdot \text{stride}, \quad j = 0, \ldots, \text{count} - 1,
\]
and
\[
B[j] = \text{blocklength}, \quad j = 0, \ldots, \text{count} - 1.
\]

\text{Hindexed} The function \text{MPI\_TYPE\_CREATE\_HINDEXED} is identical to \text{MPI\_TYPE\_INDEXED}, except that block displacements in \text{array\_of\_displacements} are specified in bytes, rather than in multiples of the \text{oldtype} extent.

\text{MPI\_TYPE\_CREATE\_HINDEXED(count, array\_of\_blocklengths, array\_of\_displacements, oldtype, newtype)}

\begin{verbatim}
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 nonnegative integers)
IN array\_of\_displacements byte displacement of each block (array of integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)
\end{verbatim}

\text{int MPI\_Type\_create\_hindexed(int count, int array\_of\_blocklengths[],
MPI\_Aint array\_of\_displacements[], MPI\_Datatype oldtype,
MPI\_Datatype \*newtype)}

\text{MPI\_TYPE\_CREATE\_HINDEXED(COUNT, ARRAY\_OF\_BLOCKLENGTHS,
ARRAY\_OF\_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)}

\text{INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI\_ADDRESS\_KIND) ARRAY\_OF\_DISPLACEMENTS(*)
\}}

\text{MPI:Datatype MPI:Datatype::Create\_hindexed(int count,
const int array\_of\_blocklengths[],
const MPI:Datatype \*array\_of\_displacements[])}

This function replaces \text{MPI\_TYPE\_HINDEXED}, whose use is deprecated. See also Chapter 15.

Assume that \text{oldtype} has type map,
\[\{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\}\].

\text{A call to \text{MPI\_TYPE\_VECTOR(count, blocklength, stride, oldtype, newtype)} is equivalent to a call to \text{MPI\_TYPE\_INDEXED(count, B, D, oldtype, newtype)} where}
\[
D[j] = j \cdot \text{stride}, \quad j = 0, \ldots, \text{count} - 1,
\]
and
\[
B[j] = \text{blocklength}, \quad j = 0, \ldots, \text{count} - 1.
\]

\text{Hindexed} The function \text{MPI\_TYPE\_CREATE\_HINDEXED} is identical to \text{MPI\_TYPE\_INDEXED}, except that block displacements in \text{array\_of\_displacements} are specified in bytes, rather than in multiples of the \text{oldtype} extent.

\text{MPI\_TYPE\_CREATE\_HINDEXED(count, array\_of\_blocklengths, array\_of\_displacements, oldtype, newtype)}

\begin{verbatim}
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 nonnegative integers)
IN array\_of\_displacements byte displacement of each block (array of integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)
\end{verbatim}

\text{int MPI\_Type\_create\_hindexed(int count, int array\_of\_blocklengths[],
MPI\_Aint array\_of\_displacements[], MPI\_Datatype oldtype,
MPI\_Datatype \*newtype)}

\text{MPI\_TYPE\_CREATE\_HINDEXED(COUNT, ARRAY\_OF\_BLOCKLENGTHS,
ARRAY\_OF\_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)}

\text{INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(*), OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI\_ADDRESS\_KIND) ARRAY\_OF\_DISPLACEMENTS(*)
\}}

\text{MPI:Datatype MPI:Datatype::Create\_hindexed(int count,
const int array\_of\_blocklengths[],
const MPI::Aint array\_of\_displacements[])}

This function replaces \text{MPI\_TYPE\_HINDEXED}, whose use is deprecated. See also Chapter 15.

Assume that \text{oldtype} has type map,
\[\{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\}\].
with extent \( ex \). Let \( B \) be the \texttt{array_of_displacements} argument and \( D \) be the \texttt{array_of_displacements} argument. The newly created datatype has a type map with \( n \cdot \sum_{i=0}^{n-1} B[i] \) entries:

\[
\{(\texttt{type0}, \texttt{disp0} + D[0]), \ldots, (\texttt{type}_{n-1}, \texttt{disp}_{n-1} + D[0]), \ldots, \\
(\texttt{type0}, \texttt{disp0} + D[0] + (B[0] - 1) \cdot ex), \ldots, \\
(\texttt{type}_{n-1}, \texttt{disp}_{n-1} + D[0] + (B[0] - 1) \cdot ex), \ldots, \\
(\texttt{type0}, \texttt{disp0} + D[count - 1], \ldots, (\texttt{type}_{n-1}, \texttt{disp}_{n-1} + D[count - 1]), \ldots, \\
(\texttt{type0}, \texttt{disp0} + D[count - 1] + (B[count - 1] - 1) \cdot ex), \ldots, \\
(\texttt{type}_{n-1}, \texttt{disp}_{n-1} + D[count - 1] + (B[count - 1] - 1) \cdot ex)\}.
\]

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

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

\begin{verbatim}
IN count length of array of displacements (non-negative integer)
IN blocklength size of block (non-negative integer)
IN array\_of\_displacements array of displacements (array of integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)
\end{verbatim}

\begin{verbatim}
int MPI\_Type\_create\_indexed\_block(int count, int blocklength, 
int array\_of\_displacements[], MPI\_Datatype oldtype, 
MPI\_Datatype *newtype)
\end{verbatim}

\texttt{MPI\_TYPE\_CREATE\_INDEXED\_BLOCK(COUNT, BLOCKLENGTH, ARRAY\_OF\_DISPLACEMENTS, 
OLDTYPE, NEWTYPE, IERROR)}

\begin{verbatim}
INTEGER COUNT, BLOCKLENGTH, ARRAY\_OF\_DISPLACEMENTS(*), OLDTYPE, 
NEWTYPE, IERROR
\end{verbatim}

\texttt{MPI::Datatype MPI::Datatype::Create\_indexed\_block(int count, 
int blocklength, const int array\_of\_displacements[])} const

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

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

\begin{verbatim}
IN count length of array of displacements (non-negative integer)
IN blocklength size of block (non-negative integer)
IN array\_of\_displacements array of displacements (array of integer)
IN oldtype old datatype (handle)
OUT newtype new datatype (handle)
\end{verbatim}

\begin{verbatim}
int MPI\_Type\_create\_indexed\_block(int count, int blocklength, 
int array\_of\_displacements[], MPI\_Datatype oldtype, 
MPI\_Datatype *newtype)
\end{verbatim}

\texttt{MPI\_TYPE\_CREATE\_INDEXED\_BLOCK(COUNT, BLOCKLENGTH, ARRAY\_OF\_DISPLACEMENTS, 
OLDTYPE, NEWTYPE, IERROR)}

\begin{verbatim}
INTEGER COUNT, BLOCKLENGTH, ARRAY\_OF\_DISPLACEMENTS(*), OLDTYPE, 
NEWTYPE, IERROR
\end{verbatim}

\texttt{MPI::Datatype MPI::Datatype::Create\_indexed\_block(int count, 
int blocklength, const int array\_of\_displacements[])} const
Struct MPI_TYPE_STRUCT is the most general type constructor. It further generalizes MPI_TYPE_CREATE_HINDEXED in that it allows each block to consist of replications of different datatypes.

MPI_TYPE_CREATE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

This function replaces MPI_TYPE_STRUCT, whose use is deprecated. See also Chapter 15.

Example 4.6 Let type1 have type map,

\[
\{(\text{double}, 0), (\text{char}, 8)\},
\]

with extent 16. Let \(B = (2, 1, 3)\), \(D = (0, 16, 26)\), and \(T = (\text{MPI_FLOAT}, \text{type1}, \text{MPI_CHAR})\).

Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,

\[
\{(\text{float}, 0), (\text{float}, 4), (\text{double}, 16), (\text{char}, 24), (\text{char}, 26), (\text{char}, 27), (\text{char}, 28)\}.
\]

That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)

Example 4.6 Let type1 have type map,

\[
\{(\text{double}, 0), (\text{char}, 8)\},
\]

with extent 16. Let \(B = (2, 1, 3)\), \(D = (0, 16, 26)\), and \(T = (\text{MPI_FLOAT}, \text{type1}, \text{MPI_CHAR})\).

Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,

\[
\{(\text{float}, 0), (\text{float}, 4), (\text{double}, 16), (\text{char}, 24), (\text{char}, 26), (\text{char}, 27), (\text{char}, 28)\}.
\]

That is, two copies of MPI_FLOAT starting at 0, followed by one copy of type1 starting at 16, followed by three copies of MPI_CHAR, starting at 26. (We assume that a float occupies four bytes.)
4.1. DERIVED DATATYPES

In general, let $T$ be the array_of_types argument, where $T[i]$ is a handle to,

type_map = \{(type_0^i, disp_0^i), ... (type_{n-1}^i, disp_{n-1}^i)\},

with extent $ex_i$. Let $B$ be the array_of_blocklength argument and $D$ be the array_of_displacements argument. Let $c$ be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{n-1} B[i] \cdot n_i$ entries:

\[
\{(type_0^i, disp_0^i + D[0]), ..., (type_{n-1}^i, disp_{n-1}^i + D[0]), \ldots, (type_0^i, disp_0^i + D[0] + (B[0] - 1) \cdot ex_0), ..., (type_{n-1}^i, disp_{n-1}^i + D[0] + (B[0] - 1) \cdot ex_0), \ldots, (type_0^i, disp_0^i + D[c - 1]) + (type_{n-1}^i, disp_{n-1}^i + D[c - 1]) \ldots, (type_0^i, disp_0^i + D[c - 1] + (B[c - 1] - 1) \cdot ex_{c-1}), \ldots, (type_{n-1}^i, disp_{n-1}^i + D[c - 1] + (B[c - 1] - 1) \cdot ex_{c-1})\}.
\]

A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of $T$ is equal to oldtype.

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN ndims number of array dimensions (positive integer)
IN array_of_sizes number of elements of type oldtype in each dimension of the full array (array of positive integers)
IN array_of_subsizes number of elements of type oldtype in each dimension of the subarray (array of positive integers)
IN array_of_starts starting coordinates of the subarray in each dimension (array of nonnegative integers)
IN order array storage order flag (state)
IN oldtype array element datatype (handle)
OUT newtype new datatype (handle)

int MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

4.1.4 DERIVED DATATYPES

In general, let $T$ be the array_of_types argument, where $T[i]$ is a handle to,

type_map = \{(type_0^i, disp_0^i), ... (type_{n-1}^i, disp_{n-1}^i)\},

with extent $ex_i$. Let $B$ be the array_of_blocklength argument and $D$ be the array_of_displacements argument. Let $c$ be the count argument. Then the newly created datatype has a type map with $\sum_{i=0}^{n-1} B[i] \cdot n_i$ entries:

\[
\{(type_0^i, disp_0^i + D[0]), ..., (type_{n-1}^i, disp_{n-1}^i + D[0]), \ldots, (type_0^i, disp_0^i + D[0] + (B[0] - 1) \cdot ex_0), ..., (type_{n-1}^i, disp_{n-1}^i + D[0] + (B[0] - 1) \cdot ex_0), \ldots, (type_0^i, disp_0^i + D[c - 1]) + (type_{n-1}^i, disp_{n-1}^i + D[c - 1]) \ldots, (type_0^i, disp_0^i + D[c - 1] + (B[c - 1] - 1) \cdot ex_{c-1}), \ldots, (type_{n-1}^i, disp_{n-1}^i + D[c - 1] + (B[c - 1] - 1) \cdot ex_{c-1})\}.
\]

A call to MPI_TYPE_CREATE_HINDEXED(count, B, D, oldtype, newtype) is equivalent to a call to MPI_TYPE_CREATE_STRUCT(count, B, D, T, newtype), where each entry of $T$ is equal to oldtype.

4.1.3 Subarray Datatype Constructor

MPI_TYPE_CREATE_SUBARRAY(ndims, array_of_sizes, array_of_subsizes, array_of_starts, order, oldtype, newtype)

IN ndims number of array dimensions (positive integer)
IN array_of_sizes number of elements of type oldtype in each dimension of the full array (array of positive integers)
IN array_of_subsizes number of elements of type oldtype in each dimension of the subarray (array of positive integers)
IN array_of_starts starting coordinates of the subarray in each dimension (array of nonnegative integers)
IN order array storage order flag (state)
IN oldtype array element datatype (handle)
OUT newtype new datatype (handle)

int MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
The subarray type constructor creates an MPI datatype describing an \( n \)-dimensional subarray of an \( n \)-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 13.1.1 on page 373.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

The \texttt{ndims} parameter specifies the number of dimensions in the full data array and gives the number of elements in \texttt{array_of_sizes\[\]}, \texttt{array_of_subsizes\[\]}, and \texttt{array_of_starts\[\]}.

The number of elements of type \texttt{oldtype} in each dimension of the \( n \)-dimensional array and the requested subarray are specified by \texttt{array_of_sizes\[\]} and \texttt{array_of_subsizes\[\]}, respectively. For any dimension \( i \), it is erroneous to specify \texttt{array_of_subsizes\[i\]} < 1 or \texttt{array_of_subsizes\[i\]} > \texttt{array_of_sizes\[i\]}.

The \texttt{array_of_starts} contains the starting coordinates of each dimension of the subarray.

Arrays are assumed to be indexed starting from zero. For any dimension \( i \), it is erroneous to specify \texttt{array_of_starts\[i\]} < 0 or \texttt{array_of_starts\[i\]} > \texttt{array_of_sizes\[i\]} - \texttt{array_of_subsizes\[i\]}.

Advice to users. In a Fortran program with arrays indexed starting from 1, if the starting coordinate of a particular dimension of the subarray is \( n \), then the entry in \texttt{array_of_starts} for that dimension is \( n-1 \). (End of advice to users.)

The \texttt{order} argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:

- \texttt{MPI\_ORDER\_C} The ordering used by C arrays, (i.e., row-major order)
- \texttt{MPI\_ORDER\_FORTRAN} The ordering used by Fortran arrays, (i.e., column-major order)

A \texttt{ndims}-dimensional subarray (newtype) with no extra padding can be defined by the function \texttt{Subarray()} as follows:

\[
\text{newtype} = \text{Subarray}(\text{ndims}, \{siz0, siz1, \ldots, siz\_ndims\_1\}, \{subsize0, subsize1, \ldots, subsize\_ndims\_1\}, \{start0, start1, \ldots, start\_ndims\_1\}, \text{oldtype})
\]

Let the typemap of \texttt{oldtype} have the form:

\[
\{(\text{type0, disp0}), (\text{type1, disp1}), \ldots, \{\text{type}_n, \text{disp}_n\}\}
\]

where \texttt{type} is a predefined MPI datatype, and \( ex \) be the extent of \texttt{oldtype}. Then we define the \texttt{Subarray()} function recursively using the following three equations. Equation 4.2 defines the base step. Equation 4.3 defines the recursion step when \texttt{order} = \texttt{MPI\_ORDER\_FORTRAN}, and Equation 4.4 defines the recursion step when \texttt{order} = \texttt{MPI\_ORDER\_C}.

The subarray type constructor creates an MPI datatype describing an \( n \)-dimensional subarray of an \( n \)-dimensional array. The subarray may be situated anywhere within the full array, and may be of any nonzero size up to the size of the larger array as long as it is confined within this array. This type constructor facilitates creating filetypes to access arrays distributed in blocks among processes to a single file that contains the global array, see MPI I/O, especially Section 13.1.1 on page 373.

This type constructor can handle arrays with an arbitrary number of dimensions and works for both C and Fortran ordered matrices (i.e., row-major or column-major). Note that a C program may use Fortran order and a Fortran program may use C order.

The \texttt{ndims} parameter specifies the number of dimensions in the full data array and gives the number of elements in \texttt{array_of_sizes\[\]}, \texttt{array_of_subsizes\[\]}, and \texttt{array_of_starts\[\]}.

The number of elements of type \texttt{oldtype} in each dimension of the \( n \)-dimensional array and the requested subarray are specified by \texttt{array_of_sizes\[\]} and \texttt{array_of_subsizes\[\]}, respectively. For any dimension \( i \), it is erroneous to specify \texttt{array_of_subsizes\[i\]} < 1 or \texttt{array_of_subsizes\[i\]} > \texttt{array_of_sizes\[i\]}.

The \texttt{array_of_starts} contains the starting coordinates of each dimension of the subarray.

Arrays are assumed to be indexed starting from zero. For any dimension \( i \), it is erroneous to specify \texttt{array_of_starts\[i\]} < 0 or \texttt{array_of_starts\[i\]} > \texttt{array_of_sizes\[i\]} - \texttt{array_of_subsizes\[i\]}.

Advice to users. In a Fortran program with arrays indexed starting from 1, if the starting coordinate of a particular dimension of the subarray is \( n \), then the entry in \texttt{array_of_starts} for that dimension is \( n-1 \). (End of advice to users.)

The \texttt{order} argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:

- \texttt{MPI\_ORDER\_C} The ordering used by C arrays, (i.e., row-major order)
- \texttt{MPI\_ORDER\_FORTRAN} The ordering used by Fortran arrays, (i.e., column-major order)
Subarray(1, [size0], [subsize0], [start0],
{(typeo, disp0), (type1, disp1), . . . , (type n−1, disp n−1)})
= {
(MPI_LB, 0),
(type0, disp0 + start0 × ex), . . . , (type n−1, disp n−1 + start0 × ex),
(type0, disp0 + (start0 + 1) × ex), . . . , (type n−1, disp n−1 + (start0 + 1) × ex), . . .
(type0, disp0 + (start0 + subsize0 − 1) × ex), . . . ,
(type n−1, disp n−1 + (start0 + subsize0 − 1) × ex),
(MPI_LB, size0 × ex)}

Subarray(n_dims, [size0, size1, . . . , size n−1],
[subsize0, subsize1, . . . , subsize n−1],
[start0, start1, . . . , start n−1], oldtype)
= Subarray(n_dims − 1, [size1, size2, . . . , size n−1],
[subsize1, subsize2, . . . , subsize n−1],
[start1, start2, . . . , start n−1],
Subarray(1, [size0], [subsize0], [start0], oldtype))

 Subarray(n_dims, [size0, size1, . . . , size n−1],
[subsize0, subsize1, . . . , subsize n−1],
[start0, start1, . . . , start n−1], oldtype)
= Subarray(n_dims − 1, [size1, size2, . . . , size n−2],
[subsize1, subsize2, . . . , subsize n−2],
[start1, start2, . . . , start n−2],
Subarray(1, [size n−1], [subsize n−1], [start n−1], oldtype))

For an example use of MPI_TYPE_CREATE_SUBARRAY in the context of I/O see Section 13.9.2.

4.1.4 Distributed Array Datatype Constructor

The distributed array type constructor supports HPF-like [30] data distributions. However, unlike
in HPF, the storage order may be specified for C arrays as well as for Fortran arrays.

Advice to users. One can create an HPF-like file view using this type constructor as
follows. Complementary filetypes are created by having every process of a group call
this constructor with identical arguments (with the exception of rank which should be
set appropriately). These filetypes (along with identical disp and etype) are then used
to define the view (via MPI_FILE_SET_VIEW), see MPI I/O, especially Section 13.1.1
on page 373 and Section 13.3 on page 385. Using this view, a collective data access
operation (with identical offsets) will yield an HPF-like distribution pattern. (End of
advice to users.)
MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, array_of_distribs, array_of_dargs, array_of_psizes, order, oldtype, newtype)

IN  size  size of process group (positive integer)
IN  rank  rank in process group (nonnegative integer)
IN  ndims  number of array dimensions as well as process grid dimensions (positive integer)
IN  array_of_gsizes  number of elements of type oldtype in each dimension of global array (array of positive integers)
IN  array_of_distribs  distribution of array in each dimension (array of state)
IN  array_of_dargs  distribution argument in each dimension (array of positive integers)
IN  array_of_psizes  size of process grid in each dimension (array of positive integers)
IN  order  array storage order flag (state)
IN  oldtype  old datatype (handle)
OUT newtype  new datatype (handle)

int MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

MPI_TYPE_CREATE_DARRAY can be used to generate the datatypes corresponding to the distribution of an ndims-dimensional array of oldtype elements onto an ndims-dimensional grid of logical processes. Unused dimensions of array_of_psizes should be set to 1. (See Example 4.7, page 93.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation \( \prod_{i=0}^{ndims-1} \text{array_of_psizes}[i] = \text{size} \) must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.

Advice to users. For both Fortran and C arrays, the ordering of processes in the process grid is assumed to be row-major. This is consistent with the ordering used in virtual Cartesian process topologies in MPI. To create such virtual process topologies, or to find the coordinates of a process in the process grid, etc., users may use the corresponding process topology functions, see Chapter 7 on page 241. (End of advice to users.)

int MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
Each dimension of the array can be distributed in one of three ways:

- **MPI_DISTRIBUTE_BLOCK** - Block distribution
- **MPI_DISTRIBUTE_CYCLIC** - Cyclic distribution
- **MPI_DISTRIBUTE_NONE** - Dimension not distributed.

The constant **MPI_DISTRIBUTE_DFLT_DARG** specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension \(i\) in which the distribution is **MPI_DISTRIBUTE_BLOCK**, it is erroneous to specify \(\text{array}_i\) or \(\text{dargs}_i\). For example, the HPF layout \(\text{ARRAY(CYCLIC}(15))\) corresponds to **MPI_DISTRIBUTE_CYCLIC** with a distribution argument of 15, and the HPF layout \(\text{ARRAY(BLOCK)}\) corresponds to **MPI_DISTRIBUTE_BLOCK** with a distribution argument of **MPI_DISTRIBUTE_DFLT_DARG**.

The order argument is used as in **MPI_TYPE_CREATE_SUBARRAY** to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are **MPI_ORDER_FORTRAN** and **MPI_ORDER_C**.

This routine creates a new MPI datatype with a typemap defined in terms of a function called “cyclic()” (see below).

Without loss of generality, it suffices to define the typemap for the **MPI_DISTRIBUTE_CYCLIC** case where **MPI_DISTRIBUTE_DFLT_DARG** is not used. **MPI_DISTRIBUTE_BLOCK** and **MPI_DISTRIBUTE_NONE** can be reduced to the **MPI_DISTRIBUTE_CYCLIC** case for dimension \(i\) as follows:

If \(\text{array}_i\) is not **MPI_DISTRIBUTE_DFLT_DARG**, then **MPI_DISTRIBUTE_BLOCK** and **MPI_DISTRIBUTE_CYCLIC** are equivalent. **MPI_DISTRIBUTE_NONE** is equivalent to **MPI_DISTRIBUTE_CYCLIC** with \(\text{array}_i\) set to \(\text{array}_i\).

Finally, **MPI_DISTRIBUTE_CYCLIC** with \(\text{array}_i\) is equivalent to **MPI_DISTRIBUTE_CYCLIC** with \(\text{array}_i\) set to 1.

For **MPI_ORDER_FORTRAN**, an ndims-dimensional distributed array (\text{newtype}) is defined by the following code fragment:

```c
oldtype[0] = oldtype;
for (i = 0; i < ndims; i++) {
    oldtype[i+1] = cyclic(array_of_dargs[i],
                        array_of_gsizes[i],
                        array_of_psizes[i],
                        oldtype[i]);
}
newtype = oldtype[ndims];
```

Each dimension of the array can be distributed in one of three ways:

- **MPI_DISTRIBUTE_BLOCK** - Block distribution
- **MPI_DISTRIBUTE_CYCLIC** - Cyclic distribution
- **MPI_DISTRIBUTE_NONE** - Dimension not distributed.

The constant **MPI_DISTRIBUTE_DFLT_DARG** specifies a default distribution argument. The distribution argument for a dimension that is not distributed is ignored. For any dimension \(i\) in which the distribution is **MPI_DISTRIBUTE_BLOCK**, it is erroneous to specify \(\text{array}_i\) or \(\text{dargs}_i\). For example, the HPF layout \(\text{ARRAY(CYCLIC}(15))\) corresponds to **MPI_DISTRIBUTE_CYCLIC** with a distribution argument of 15, and the HPF layout \(\text{ARRAY(BLOCK)}\) corresponds to **MPI_DISTRIBUTE_BLOCK** with a distribution argument of **MPI_DISTRIBUTE_DFLT_DARG**.

The order argument is used as in **MPI_TYPE_CREATE_SUBARRAY** to specify the storage order. Therefore, arrays described by this type constructor may be stored in Fortran (column-major) or C (row-major) order. Valid values for order are **MPI_ORDER_FORTRAN** and **MPI_ORDER_C**.

This routine creates a new MPI datatype with a typemap defined in terms of a function called “cyclic()” (see below).

Without loss of generality, it suffices to define the typemap for the **MPI_DISTRIBUTE_CYCLIC** case where **MPI_DISTRIBUTE_DFLT_DARG** is not used. **MPI_DISTRIBUTE_BLOCK** and **MPI_DISTRIBUTE_NONE** can be reduced to the **MPI_DISTRIBUTE_CYCLIC** case for dimension \(i\) as follows:

If \(\text{array}_i\) is not **MPI_DISTRIBUTE_DFLT_DARG**, then **MPI_DISTRIBUTE_BLOCK** and **MPI_DISTRIBUTE_CYCLIC** are equivalent. **MPI_DISTRIBUTE_NONE** is equivalent to **MPI_DISTRIBUTE_CYCLIC** with \(\text{array}_i\) set to \(\text{array}_i\).

Finally, **MPI_DISTRIBUTE_CYCLIC** with \(\text{array}_i\) is equivalent to **MPI_DISTRIBUTE_CYCLIC** with \(\text{array}_i\) set to 1.

For **MPI_ORDER_FORTRAN**, an ndims-dimensional distributed array (\text{newtype}) is defined by the following code fragment:

```c
oldtype[0] = oldtype;
for (i = 0; i < ndims; i++) {
    oldtype[i+1] = cyclic(array_of_dargs[i],
                        array_of_gsizes[i],
                        array_of_psizes[i],
                        oldtype[i]);
}
newtype = oldtype[ndims];
```
For MPI_ORDER_C, the code is:

```c
oldtype[0] = oldtype;
for ( i = 0; i < ndims; i++ ) {
    oldtype[i + 1] = cyclic(array_of_dargs[ndims - i - 1],
                          array_of_gsizes[ndims - i - 1],
                          r[ndims - i - 1],
                          array_of_psizes[ndims - i - 1],
                          oldtype[i]);
}
newtype = oldtype[ndims];
```

where \( r[i] \) is the position of the process (with rank \( rank \)) in the process grid at dimension \( i \).

The values of \( r[i] \) are given by the following code fragment:

```c
t_rank = rank;
t_size = 1;
for ( i = 0; i < ndims; i++ )
    t_size = t_size / array_of_psizes[i];
r[i] = t_rank / t_size;
t_rank = t_rank % t_size;
}
```

Let the typemap of \( oldtype \) have the form:

\[
\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_n-1, disp_{n-1})\}
\]

where \( type_i \) is a predefined MPI datatype, and let \( ex \) be the extent of \( oldtype \).

Given the above, the function cyclic() is defined as follows:

```c
cyclic(darg, gsize, r, psize, oldtype) =
    \{(MPI_LB, 0),
    (type_0, disp_0 + r \times darg \times ex), \ldots,
    (type_0, disp_0 + r \times darg + 1 \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg - 1 \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg \times ex), \ldots,
    (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,
    (type_0, disp_0 + r \times darg + 1 \times ex + psize \times darg \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg + 1 \times ex + psize \times darg \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg \times ex + psize \times darg \times ex), \ldots\}
```

where \( type_i \) is a predefined MPI datatype, and let \( ex \) be the extent of \( oldtype \).

Given the above, the function cyclic() is defined as follows:

```c
cyclic(darg, gsize, r, psize, oldtype) =
    \{(MPI_LB, 0),
    (type_0, disp_0 + r \times darg \times ex), \ldots,
    (type_0, disp_0 + r \times darg + 1 \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg - 1 \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg \times ex), \ldots,
    (type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots,
    (type_0, disp_0 + r \times darg + 1 \times ex + psize \times darg \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg + 1 \times ex + psize \times darg \times ex), \ldots,
    (type_0, disp_0 + (r + 1) \times darg \times ex + psize \times darg \times ex), \ldots\}
```

where \( r[i] \) is the position of the process (with rank \( rank \)) in the process grid at dimension \( i \).

The values of \( r[i] \) are given by the following code fragment:

```c
t_rank = rank;
t_size = 1;
for ( i = 0; i < ndims; i++ )
    t_size = t_size / array_of_psizes[i];
r[i] = t_rank / t_size;
t_rank = t_rank % t_size;
}
```
Consider generating the filetypes corresponding to the HPF distribution:

```
<oldtype> FILEARRAY(100, 200, 300)
```

```
MPIUB gsize * ex)
```

where count is defined by this code fragment:

```
if (nblocks == 0)
    count = nblocks / psizes;
```

```
if (r < left_over)
    count = count + 1;
```

Finally, darg_last is defined by this code fragment:

```
darg_last = darg;
```

```
if (darg_last > darg)
    darg_last = darg;
else
    darg_last = num_in_last_cyclic - darg * r;
```

```
darg_last = darg;
```

Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:

```
<oldtype> FILEARRAY(100, 200, 300)
```

```
MPIUB gsize * ex)
```

where count is defined by this code fragment:

```
if (nblocks == 0)
    count = nblocks / psizes;
```

```
if (r < left_over)
    count = count + 1;
```

Finally, darg_last is defined by this code fragment:

```
darg_last = darg;
```

```
if (darg_last > darg)
    darg_last = darg;
else
    darg_last = num_in_last_cyclic - darg * r;
```

```
darg_last = darg;
```

Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:

```
<oldtype> FILEARRAY(100, 200, 300)
```

```
MPIUB gsize * ex)
```

where count is defined by this code fragment:

```
if (nblocks == 0)
    count = nblocks / psizes;
```

```
if (r < left_over)
    count = count + 1;
```

Finally, darg_last is defined by this code fragment:

```
darg_last = darg;
```

```
if (darg_last > darg)
    darg_last = darg;
else
    darg_last = num_in_last_cyclic - darg * r;
```

```
darg_last = darg;
```

Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:

```
<oldtype> FILEARRAY(100, 200, 300)
```

```
MPIUB gsize * ex)
```

where count is defined by this code fragment:

```
if (nblocks == 0)
    count = nblocks / psizes;
```

```
if (r < left_over)
    count = count + 1;
```

Finally, darg_last is defined by this code fragment:

```
darg_last = darg;
```

```
if (darg_last > darg)
    darg_last = darg;
else
    darg_last = num_in_last_cyclic - darg * r;
```

```
darg_last = darg;
```

Example 4.7 Consider generating the filetypes corresponding to the HPF distribution:

```
<oldtype> FILEARRAY(100, 200, 300)
```

```
MPIUB gsize * ex)
```

where count is defined by this code fragment:

```
if (nblocks == 0)
    count = nblocks / psizes;
```

```
if (r < left_over)
    count = count + 1;
```

Finally, darg_last is defined by this code fragment:

```
darg_last = darg;
```

```
if (darg_last > darg)
    darg_last = darg;
else
    darg_last = num_in_last_cyclic - darg * r;
```

```
darg_last = darg;
```
4.1.5 Address and Size Functions

The displacements in a general datatype are relative to some initial buffer address. Absolute addresses can be substituted for these displacements: we treat them as displacements relative to "address zero," the start of the address space. This initial address zero is indicated by the constant MPI_BOTTOM. Thus, a datatype can specify the absolute address of the entries in the communication buffer, in which case the buf argument is passed the value MPI_BOTTOM.

The address of a location in memory can be found by invoking the function

\texttt{MPI\_GET\_ADDRESS(location, address)}

\texttt{MPI\_GET\_ADDRESS(location, address)}

\texttt{IN location location in caller memory (choice)}

\texttt{OUT address address of location (integer)}

\texttt{int MPI\_Get\_address(void *location, MPI\_Aint *address)}

This function replaces MPI\_ADDRESS, whose use is deprecated. See also Chapter 15. Returns the (byte) address of location.

\texttt{Advice to users.} Current Fortran MPI codes will run unmodified, and will port to any system. However, they may fail if addresses larger than $2^{32} - 1$ are used in the program. New codes should be written so that they use the new functions.
4.1. DERIVED DATATYPES

This provides compatibility with C/C++ and avoids errors on 64 bit architectures. However, such newly written codes may need to be (slightly) rewritten to port to old Fortran 77 environments that do not support KIND declarations. (End of advice to users.)

Example 4.8 Using MPI_GET_ADDRESS for an array.

```c
REAL A(100,100)
INTEGER(KIND=MPI_ADDRESS_KIND) I1, I2, DIFF
CALL MPI_GET_ADDRESS(A(1,1), I1, IERROR)
CALL MPI_GET_ADDRESS(A(10,10), I2, IERROR)
DIFF = I2 - I1
! The value of DIFF is 909*sizeofreal; the values of I1 and I2 are
! implementation dependent.
```

Advice to users. C users may be tempted to avoid the usage of MPI_GET_ADDRESS and rely on the availability of the address operator &. Note, however, that & cast-expression is a pointer, not an address. ISO C does not require that the value of a pointer (or the pointer cast to int) be the absolute address of the object pointed at — although this is commonly the case. Furthermore, referencing may not have a unique definition on machines with a segmented address space. The use of MPI_GET_ADDRESS to “reference” C variables guarantees portability to such machines as well. (End of advice to users.)

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 on pages 463 and 466. (End of advice to users.)

The following auxiliary function provides useful information on derived datatypes.

```c
MPI_TYPE_SIZE(datatype, size)
IN    datatype   datatype (handle)
OUT   size       datatype size (integer)
```

```c
int MPI_Type_size(MPI_Datatype datatype, int *size)
MPL_TYPE_SIZE(DATATYPE, SIZE, IERROR)
INTEGER DATATYPE, SIZE, IERROR
```

```c
int MPI::Datatype::Get_size() const
MPL_TYPE_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.
4.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 96. This allows one to define a datatype that has “holes” at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. Also, the user may want to override the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to override default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional “pseudo-datatypes,” MPI LB and MPI UB, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space (extent(MPI LB) = extent(MPI UB) = 0). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example 4.9 Let D = (-3, 0, 6); T = (MPI LB, MPI INT, MPI UB), and B = (1, 1, 1). Then a call to MPI TYPE STRUCT(3, B, D, T, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence \{(lb, -3), (int, 0), (ub, 6)\}. If this type is replicated twice by a call to MPI TYPE CONTIGUOUS(2, type1, type2) then the newly created type can be described by the sequence \{(lb, -3), (int, 0), (int, 9), (ub, 15)\}. (An entry of type ub can be deleted if there is another entry of type ub with a higher displacement; an entry of type lb can be deleted if there is another entry of type lb with a lower displacement.)

In general, if

\[\text{Typemap} = \{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\},\]

then the lower bound of Typemap is defined to be

\[\text{lb}(\text{Typemap}) = \begin{cases} \min_j \text{disp}_j & \text{if no entry has basic type lb} \\ \min_j \{\text{disp}_j \text{ such that type}_j = \text{lb}\} & \text{otherwise}\end{cases}\]

Similarly, the upper bound of Typemap is defined to be

\[\text{ub}(\text{Typemap}) = \begin{cases} \max_j \text{disp}_j + \text{sizeof}(\text{type}_j) + \epsilon & \text{if no entry has basic type ub} \\ \max_j \{\text{disp}_j \text{ such that type}_j = \text{ub}\} & \text{otherwise}\end{cases}\]

Then

\[\text{extent}(\text{Typemap}) = \text{ub}(\text{Typemap}) - \text{lb}(\text{Typemap})\]

If type\(_i\) requires alignment to a byte address that is a multiple of \(k_i\), then \(\epsilon\) is the least nonnegative increment needed to round extent(Typemap) to the next multiple of max\(_i\) \(k_i\).

The formal definitions given for the various datatype constructors apply now, with the amended definition of extent.

4.1.6 Lower-Bound and Upper-Bound Markers

It is often convenient to define explicitly the lower bound and upper bound of a type map, and override the definition given on page 96. This allows one to define a datatype that has “holes” at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound. Examples of such usage are provided in Section 4.1.14. Also, the user may want to override the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to override default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional “pseudo-datatypes,” MPI LB and MPI UB, that can be used, respectively, to mark the lower bound or the upper bound of a datatype. These pseudo-datatypes occupy no space (extent(MPI LB) = extent(MPI UB) = 0). They do not affect the size or count of a datatype, and do not affect the content of a message created with this datatype. However, they do affect the definition of the extent of a datatype and, therefore, affect the outcome of a replication of this datatype by a datatype constructor.

Example 4.9 Let D = (-3, 0, 6); T = (MPI LB, MPI INT, MPI UB), and B = (1, 1, 1). Then a call to MPI TYPE STRUCT(3, B, D, T, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the sequence \{(lb, -3), (int, 0), (int, 9), (ub, 6)\}. If this type is replicated twice by a call to MPI TYPE CONTIGUOUS(2, type1, type2) then the newly created type can be described by the sequence \{(lb, -3), (int, 0), (int, 9), (ub, 15)\}. (An entry of type ub can be deleted if there is another entry of type ub with a higher displacement; an entry of type lb can be deleted if there is another entry of type lb with a lower displacement.)

In general, if

\[\text{Typemap} = \{(\text{type}_0, \text{disp}_0), \ldots, (\text{type}_{n-1}, \text{disp}_{n-1})\},\]

then the lower bound of Typemap is defined to be

\[\text{lb}(\text{Typemap}) = \begin{cases} \min_j \text{disp}_j & \text{if no entry has basic type lb} \\ \min_j \{\text{disp}_j \text{ such that type}_j = \text{lb}\} & \text{otherwise}\end{cases}\]

Similarly, the upper bound of Typemap is defined to be

\[\text{ub}(\text{Typemap}) = \begin{cases} \max_j \text{disp}_j + \text{sizeof}(\text{type}_j) + \epsilon & \text{if no entry has basic type ub} \\ \max_j \{\text{disp}_j \text{ such that type}_j = \text{ub}\} & \text{otherwise}\end{cases}\]

Then

\[\text{extent}(\text{Typemap}) = \text{ub}(\text{Typemap}) - \text{lb}(\text{Typemap})\]

If type\(_i\) requires alignment to a byte address that is a multiple of \(k_i\), then \(\epsilon\) is the least nonnegative increment needed to round extent(Typemap) to the next multiple of max\(_i\) \(k_i\).

The formal definitions given for the various datatype constructors apply now, with the amended definition of extent.
4.1. DERIVED DATATYPES

4.1.7 Extent and Bounds of Datatypes

The following function replaces the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT. It also returns address sized integers, in the Fortran binding. The use of MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT is deprecated.

MPI_TYPE_GET_EXTENT(datatype, lb, extent)
IN  datatype  datatype to get information on (handle)
OUT lb  lower bound of datatype (integer)
OUT extent  extent of datatype (integer)

int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)

MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
INTEGER DATATYPE, IERROR
INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT

void MPI::Datatype::Get_extent(MPI::Aint& lb, MPI::Aint& extent) const

Returns the lower bound and the extent of datatype (as defined in Section 4.1.6 on page 96).

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers (MPI_LB and MPI_UB). This is useful, as it allows to control the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the count argument in a send or receive call. However, the current mechanism for achieving it is painful; also it is restrictive. MPI_LB and MPI_UB are "sticky": once present in a datatype, they cannot be overridden (e.g., the upper bound can be moved up, by adding a new MPI_UB marker, but cannot be moved down below an existing MPI_UB marker). A new type constructor is provided to facilitate these changes. The use of MPI_LB and MPI_UB is deprecated.

MPI_TYPE_CREATE_RESIZED(oldtype, lb, extent, newtype)
IN  oldtype  input datatype (handle)
IN lb  new lower bound of datatype (integer)
IN extent  new extent of datatype (integer)
OUT newtype  output datatype (handle)

int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype)

MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
INTEGER OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND = MPI_ADDRESS_KIND) LB, EXTENT

void MPI::Datatype::Create_resized(const MPI::Aint lb, const MPI::Aint extent) const

MPI::Datatype MPI::Datatype::Create_resized(const MPI::Aint lb, const MPI::Aint extent) const

The following function replaces the three functions MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT. It also returns address sized integers, in the Fortran binding. The use of MPI_TYPE_UB, MPI_TYPE_LB and MPI_TYPE_EXTENT is deprecated.
Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be ub + extent. Any previous lb and ub markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and ub arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

Advice to users. It is strongly recommended that users use these two new functions, rather than the old MPI-1 functions to set and access lower bound, upper bound and extent of datatypes. (End of advice to users.)

4.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 5.5 on page 137) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent using the MPI_LB and MPI UB values. A function is provided which returns the true extent of the datatype.

```
MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)
```

IN  datatype  datatype to get information on (handle)

OUT true_lb  true lower bound of datatype (integer)

OUT true_extent  true size of datatype (integer)

```
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)
```

MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE LB, TRUE EXTENT, IERROR)
INTEGER DATATYPE, IERROR

```
void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb, MPI::Aint& true_extent) const
```

true_lb returns the offset of the lowest unit of storage which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring MPI LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI LB and MPI UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

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

Then

```
true_lb(Typemap) = \min_j\{disp_j : type_j \neq lb, ub\},
true_ub(Typemap) = \max_j\{disp_j + sizeof(type_j) : type_j \neq lb, ub\},
```

Returns in newtype a handle to a new datatype that is identical to oldtype, except that the lower bound of this new datatype is set to be lb, and its upper bound is set to be lb + extent. Any previous lb and ub markers are erased, and a new pair of lower bound and upper bound markers are put in the positions indicated by the lb and ub arguments. This affects the behavior of the datatype when used in communication operations, with count > 1, and when used in the construction of new derived datatypes.

Advice to users. It is strongly recommended that users use these two new functions, rather than the old MPI-1 functions to set and access lower bound, upper bound and extent of datatypes. (End of advice to users.)

4.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 5.5 on page 137) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent using the MPI_LB and MPI UB values. A function is provided which returns the true extent of the datatype.

```
MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)
```

IN  datatype  datatype to get information on (handle)

OUT true_lb  true lower bound of datatype (integer)

OUT true_extent  true size of datatype (integer)

```
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb, MPI_Aint *true_extent)
```

MPI_TYPE_GET_TRUE_EXTENT(DATATYPE, TRUE LB, TRUE EXTENT, IERROR)
INTEGER(KIND = MPI_ADDRESS_KIND) TRUE LB, TRUE EXTENT

```
void MPI::Datatype::Get_true_extent(MPI::Aint& true_lb, MPI::Aint& true_extent) const
```

true_lb returns the offset of the lowest unit of storage which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring MPI LB markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring MPI LB and MPI UB markers, and performing no rounding for alignment. If the typemap associated with datatype is

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

Then

```
true_lb(Typemap) = \min_j\{disp_j : type_j \neq lb, ub\},
true_ub(Typemap) = \max_j\{disp_j + sizeof(type_j) : type_j \neq lb, ub\},
```

4.1. DERIVED DATATYPES

and

\[ \text{true}_\text{extent}(\text{typemap}) = \text{true}_\text{ub}(\text{typemap}) - \text{true}_\text{lb}(\text{typemap}). \]

(Readers should compare this with the definitions in Section 4.1.6 on page 96 and Section 4.1.7 on page 97, which describe the function MPI\_TYPE\_GET\_EXTENT.)

The true\_extent is the minimum number of bytes of memory necessary to hold a datatype, uncompressed.

4.1.9 Commit and Free

A datatype object has to be committed before it can be used in a communication. As an argument in datatype constructors, uncommitted and also committed datatypes can be used. There is no need to commit basic datatypes. They are “pre-committed.”

\[ \text{MPI\_TYPE\_COMMIT}(\text{datatype}) \]

\[ \text{INOUT} \quad \text{datatype that is committed (handle)} \]

void MPI::Datatype::Commit()

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

\textit{Advice to implementors.} The system may “compile” at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (\textit{End of advice to implementors.})

\[ \text{MPI\_TYPE\_COMMIT}(\text{datatype}) \]

\[ \text{INOUT} \quad \text{datatype that is committed (handle)} \]

int MPI_Type_commit(MPI_Datatype *datatype)

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

\[ \text{MPI\_TYPE\_COMMIT}(\text{DATATYPE}, \text{IERROR}) \]

INTEGER DATATYPE, IERROR

void MPI::Datatype::Commit()

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

\textit{Advice to implementors.} The system may “compile” at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (\textit{End of advice to implementors.})

\[ \text{MPI\_TYPE\_COMMIT}(\text{datatype}) \]

\[ \text{INOUT} \quad \text{datatype that is committed (handle)} \]

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

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

INTEGER DATATYPE, IERROR

void MPI::Datatype::Commit()

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

\textit{Advice to implementors.} The system may “compile” at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (\textit{End of advice to implementors.})

\[ \text{MPI\_TYPE\_COMMIT}(\text{datatype}) \]

\[ \text{INOUT} \quad \text{datatype that is committed (handle)} \]

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

INT INTEGER DATATYPE, IERROR

void MPI::Datatype::Commit()

The commit operation commits the datatype, that is, the formal description of a communication buffer, not the content of that buffer. Thus, after a datatype has been committed, it can be repeatedly reused to communicate the changing content of a buffer or, indeed, the content of different buffers, with different starting addresses.

\textit{Advice to implementors.} The system may “compile” at commit time an internal representation for the datatype that facilitates communication, e.g. change from a compacted representation to a flat representation of the datatype, and select the most convenient transfer mechanism. (\textit{End of advice to implementors.})

\[ \text{MPI\_TYPE\_COMMIT}(\text{datatype}) \]

\[ \text{INOUT} \quad \text{datatype that is committed (handle)} \]

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

INT INTEGER DATATYPE, IERROR

void MPI::Datatype::Commit()
CALL MPI_TYPE_COMMIT(type1, ierr)

! now type1 can be used anew for communication

MPI_TYPE_FREE(datatype)

INOUT datatype
datatype that is freed (handle)

int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE(DATATYPE, IERROR)

INTEGER DATATYPE, IERROR

void MPI::Datatype::Free()

Marks the datatype object associated with datatype for deallocation and sets datatype to MPI_DATATYPE_NULL. Any communication that is currently using this datatype will complete normally. Freeing a datatype does not affect any other datatype that was built from the freed datatype. The system behaves as if input datatype arguments to derived datatype constructors are passed by value.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather than copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (End of advice to implementors.)

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP(type, newtype)

IN type
datatype (handle)

OUT newtype
copy of type (handle)

int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)

MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)

INTEGER TYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Dup() const

MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as type and any copied cached information, see Section 6.7.4 on page 230. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded.

Advice to implementors. The implementation may keep a reference count of active communications that use the datatype, in order to decide when to free it. Also, one may implement constructors of derived datatypes so that they keep pointers to their datatype arguments, rather than copying them. In this case, one needs to keep track of active datatype definition references in order to know when a datatype object can be freed. (End of advice to implementors.)

4.1.10 Duplicating a Datatype

MPI_TYPE_DUP(type, newtype)

IN type
datatype (handle)

OUT newtype
copy of type (handle)

int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)

MPI_TYPE_DUP(TYPE, NEWTYPE, IERROR)

INTEGER TYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Dup() const

MPI_TYPE_DUP is a type constructor which duplicates the existing type with associated key values. For each key value, the respective copy callback function determines the attribute value associated with this key in the new communicator; one particular action that a copy callback may take is to delete the attribute from the new datatype. Returns in newtype a new datatype with exactly the same properties as type and any copied cached information, see Section 6.7.4 on page 230. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded.
with the functions in Section 4.1.13. The newtype has the same committed state as the old type.

4.1.11 Use of General Datatypes in Communication

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

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

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

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

```
{(type0, disp0), ..., (typen-1, dispn-1)},
```

and extent `extent`. (Empty entries of "pseudo-type" `MPI_UB` and `MPI_LB` are not listed in the type map, but they affect the value of `extent`.) The send operation sends `n · count` entries, where entry `i · n + j` is at location `addr_ij = buf + extent · i + dispj`, and has type `type_j`, for `i = 0, ..., count - 1` and `j = 0, ..., n - 1`. These entries need not be contiguous, nor distinct; their order can be arbitrary.

The variable stored at address `addr_ij` in the calling program should be of a type that matches `type_j`, where type matching is defined as in Section 3.3.1. The message sent contains `n · count` entries, where entry `i · n + j` has type `type_j`.

Similarly, suppose that a receive operation `MPI_RECV(buf, count, datatype, source, tag, comm)`, where `datatype` has type map,

```
{(type0, disp0), ..., (typen-1, dispn-1)},
```

with extent `extent`. (Again, empty entries of "pseudo-type" `MPI_UB` and `MPI_LB` are not listed in the type map, but they affect the value of `extent`.) This receive operation receives `n · count` entries, where entry `i · n + j` is at location `buf + extent · i + dispj` and has type `type_j`. If the incoming message consists of `k` elements, then we must have `k ≤ n · count`; the `i · n + j`-th element of the message should have a type that matches `type_j`.

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

Example 4.11 This example shows that type matching is defined in terms of the basic types that a derived type consists of.

```
CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
```
CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)  
...  
CALL MPI_SEND( a, 4, MPI_REAL, ...)  
CALL MPI_SEND( a, 2, type2, ...)  
CALL MPI_SEND( a, 1, type22, ...)  
CALL MPI_SEND( a, 1, type4, ...)  
...  
CALL MPI_RECV( a, 4, MPI_REAL, ...)  
CALL MPI_RECV( a, 2, type2, ...)  
CALL MPI_RECV( a, 1, type22, ...)  
CALL MPI_RECV( a, 1, type4, ...)  

Each of the sends matches any of the receives.

A datatype may specify overlapping entries. The use of such a datatype in a receive operation is erroneous. (This is erroneous even if the actual message received is short enough not to write any entry more than once.)

Suppose that MPI_RECV(buf, count, datatype, dest, tag, comm, status) is executed, where datatype has type map,

\{
(type, disp), ..., (type_{n-1}, disp_{n-1})\}

The received message need not fill all the receive buffer, nor does it need to fill a number of locations which is a multiple of \(n\). Any number, \(k\), of basic elements can be received, where \(0 \leq k \leq \text{count} \cdot n\). The number of basic elements received can be retrieved from status using the function MPI_GET_ELEMENTS.

\[
\text{MPI\_GET\_ELEMENTS}(\text{status}, \text{datatype}, \text{count})
\]

IN status return status of receive operation (Status)
IN datatype datatype used by receive operation (handle)
OUT count number of received basic elements (integer)

\[
\text{int MPI\_Get\_elements(MPI\_Status \ast \text{status}, MPI\_Datatype \text{datatype}, int \ast \text{count})}
\]

\[
\text{INTEGER STATUS(MPI\_STATUS\_SIZE), DATATYPE, COUNT, IERROR}
\]

The previously defined function, MPI_GET_COUNT (Section 3.2.5), has a different behavior. It returns the number of “top-level entries” received, i.e. the number of “copies” of type datatype. In the previous example, MPI_GET_COUNT may return any integer value \(k\), where \(0 \leq k \leq \text{count}\). If MPI_GET_COUNT returns \(k\), then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS) is \(n \cdot k\). If the number of basic elements received is not a multiple of \(n\), that is, if the receive operation has not received an integral number of datatype “copies,” then MPI_GET_COUNT returns the value MPI_UNDEFINED. The datatype argument should match the argument provided by the receive call that set the status variable.
4.1. DERIVED DATATYPES

Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

```fortran
Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)

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

The function MPI_GET_ELEMENTS can also be used after a probe to find the number of elements in the probed message. Note that the two functions MPI_GET_COUNT and MPI_GET_ELEMENTS return the same values when they are used with basic datatypes.

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

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

4.1.12 Correct Use of Addresses

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

Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

```fortran
Example 4.12 Usage of MPI_GET_COUNT and MPI_GET_ELEMENTS.

CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
CALL MPI_TYPE_COMMIT(Type2, ierr)

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

The function MPI_GET_ELEMENTS can also be used after a probe to find the number of elements in the probed message. Note that the two functions MPI_GET_COUNT and MPI_GET_ELEMENTS return the same values when they are used with basic datatypes.

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

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

4.1.12 Correct Use of Addresses

Successively declared variables in C or Fortran are not necessarily stored at contiguous locations. Thus, care must be exercised that displacements do not cross from one variable to another. Also, in machines with a segmented address space, addresses are not unique and address arithmetic has some peculiar properties. Thus, the use of addresses, that is, displacements relative to the start address MPI_BOTTOM, has to be restricted.
Variables belong to the same sequential storage if they belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Valid addresses are defined recursively as follows:

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

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

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

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

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

4.1.13 Decoding a Datatype

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

4.1.13 Decoding a Datatype

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

**MPI_TYPE_GET_ENVELOPE**

<table>
<thead>
<tr>
<th>IN</th>
<th>datatype</th>
<th>datatype to access (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>num_integers</td>
<td>number of input integers used in the call constructing combiner (nonnegative integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>num_addresses</td>
<td>number of input addresses used in the call constructing combiner (nonnegative integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>num_datatypes</td>
<td>number of input datatypes used in the call constructing combiner (nonnegative integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>combiner</td>
<td>combiner (state)</td>
</tr>
</tbody>
</table>

```c
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, int *num_addresses, int *num_datatypes, int *combiner)
```

**MPI_TYPE_GET_ENVELOPE**

```c
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTETERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR)
```

**INTEGER DATATYPE, NUM_INTETERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER, IERROR**

```c
void MPI::Datatype::Get_envelope(int &num_integers, int &num_addresses, int &num_datatypes, int &combiner) const
```

For the given datatype, **MPI_TYPE_GET_ENVELOPE** returns information on the number and type of input arguments used in the call that created the datatype. The number-of-arguments values returned can be used to provide sufficiently large arrays in the decoding routine **MPI_TYPE_GET_CONTENTS**. This call and the meaning of the returned values is described below. The combiner reflects the MPI datatype constructor call that was used in creating datatype.

**Rationale.** By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the same as another when the information obtained from **MPI_TYPE_GET_CONTENTS** may be used with either to produce the same outcome. C calls **MPI_Type_hindexed** and **MPI_Type_create_hindexed** are always effectively the same while the Fortran call **MPI_Type_HINDEXED** will be different than either of these in some MPI implementations. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

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

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

If combiner is **MPI_COMBINER_NAMED** then datatype is a named predefined datatype.

**Rationale.** By requiring that the combiner reflect the constructor used in the creation of the datatype, the decoded information can be used to effectively recreate the calling sequence used in the original creation. One call is effectively the same as another when the information obtained from **MPI_TYPE_GET_CONTENTS** may be used with either to produce the same outcome. C calls **MPI_Type_hindexed** and **MPI_Type_create_hindexed** are always effectively the same while the Fortran call **MPI_Type_HINDEXED** will be different than either of these in some MPI implementations. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

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

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

If combiner is **MPI_COMBINER_NAMED** then datatype is a named predefined datatype.
For deprecated calls with address arguments, we sometimes need to differentiate whether the call used an integer or an address size argument. For example, there are two combin-ers for hvector: MPI_COMBINER_HVECTOR_INTEGER and MPI_COMBINER_HVECTOR. The former is used if it was the MPI-1 call from Fortran, and the latter is used if it was the MPI-1 call from C or C++. However, on systems where MPI_ADDRESS_KIND = MPI_INTEGER_KIND (i.e., where integer arguments and address size arguments are the same), the combiner MPI_COMBINER_HVECTOR may be returned for a datatype constructed by a call to MPI_TYPE_HVECTOR from Fortran. Similarly, MPI_COMBINER_HINDEXED may be returned for a datatype constructed by a call to MPI_TYPE_HINDEXED from Fortran, and MPI_COMBINER_STRUCT may be returned for a datatype constructed by a call to MPI_TYPE_STRUCT from Fortran. On such systems, one need not differentiate constructors that take address size arguments from constructors that take integer arguments, since these are the same. The preferred calls all use address sized arguments so two combin-ers are not required for them.

**Rationale.** For recreating the original call, it is important to know if address informa-tion may have been truncated. The deprecated calls from Fortran for a few routines could be subject to truncation in the case where the default INTEGER size is smaller than the size of an address. (End of rationale.)
The actual arguments used in the creation call for a datatype can be obtained from the call:

```
MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes)
```

<table>
<thead>
<tr>
<th>IN</th>
<th>datatype</th>
<th>datatype to access (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>max_integers</td>
<td>number of elements in array_of_integers (nonnegative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>max_addresses</td>
<td>number of elements in array_of_addresses (nonnegative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>max_datatypes</td>
<td>number of elements in array_of_datatypes (nonnegative integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>array_of_integers</td>
<td>contains integer arguments used in constructing datatype (array of integers)</td>
</tr>
<tr>
<td>OUT</td>
<td>array_of_addresses</td>
<td>contains address arguments used in constructing datatype (array of integers)</td>
</tr>
<tr>
<td>OUT</td>
<td>array_of_datatypes</td>
<td>contains datatype arguments used in constructing datatype (array of handles)</td>
</tr>
</tbody>
</table>

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers, int max_addresses, int max_datatypes, MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[])
```

The values given for max_integers, max_addresses, and max_datatypes must be at least as large as the value returned in num_integers, num_addresses, and num_datatypes, respectively, in the call MPI_Type_GET_ENVELOPE for the same datatype argument.

Rationale. The arguments max_integers, max_addresses, and max_datatypes allow for error checking in the call. (End of rationale.)

The datatypes returned in array_of_datatypes are handles to datatype objects that are equivalent to the datatypes used in the original construction call. If these were derived
datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the == or .EQ. comparison operator to determine the datatype involved. (End of rationale.)

Advice to implementors. The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (End of advice to implementors.)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (End of rationale.)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type INTEGER. In the preferred calls, the address arguments are of type INTEGER(KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type INTEGER(KIND=MPI_ADDRESS_KIND) will be at least as large as the INTEGER argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (End of rationale.)

The following defines what values are placed in each entry of the returned arrays depending on the datatype constructor used for datatype. It also specifies the size of the arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, the following calls were made:

datatypes, then the returned datatypes are new datatype objects, and the user is responsible for freeing these datatypes with MPI_TYPE_FREE. If these were predefined datatypes, then the returned datatype is equal to that (constant) predefined datatype and cannot be freed.

The committed state of returned derived datatypes is undefined, i.e., the datatypes may or may not be committed. Furthermore, the content of attributes of returned datatypes is undefined.

Note that MPI_TYPE_GET_CONTENTS can be invoked with a datatype argument that was constructed using MPI_TYPE_CREATE_F90_REAL, MPI_TYPE_CREATE_F90_INTEGER, or MPI_TYPE_CREATE_F90_COMPLEX (an unnamed predefined datatype). In such a case, an empty array_of_datatypes is returned.

Rationale. The definition of datatype equivalence implies that equivalent predefined datatypes are equal. By requiring the same handle for named predefined datatypes, it is possible to use the == or .EQ. comparison operator to determine the datatype involved. (End of rationale.)

Advice to implementors. The datatypes returned in array_of_datatypes must appear to the user as if each is an equivalent copy of the datatype used in the type constructor call. Whether this is done by creating a new datatype or via another mechanism such as a reference count mechanism is up to the implementation as long as the semantics are preserved. (End of advice to implementors.)

Rationale. The committed state and attributes of the returned datatype is deliberately left vague. The datatype used in the original construction may have been modified since its use in the constructor call. Attributes can be added, removed, or modified as well as having the datatype committed. The semantics given allow for a reference count implementation without having to track these changes. (End of rationale.)

In the deprecated datatype constructor calls, the address arguments in Fortran are of type INTEGER. In the preferred calls, the address arguments are of type INTEGER(KIND=MPI_ADDRESS_KIND). The call MPI_TYPE_GET_CONTENTS returns all addresses in an argument of type INTEGER(KIND=MPI_ADDRESS_KIND). This is true even if the deprecated calls were used. Thus, the location of values returned can be thought of as being returned by the C bindings. It can also be determined by examining the preferred calls for datatype constructors for the deprecated calls that involve addresses.

Rationale. By having all address arguments returned in the array_of_addresses argument, the result from a C and Fortran decoding of a datatype gives the result in the same argument. It is assumed that an integer of type INTEGER(KIND=MPI_ADDRESS_KIND) will be at least as large as the INTEGER argument used in datatype construction with the old MPI-1 calls so no loss of information will occur. (End of rationale.)

The following defines what values are placed in each entry of the returned arrays depending on the datatype constructor used for datatype. It also specifies the size of the arrays needed which is the values returned by MPI_TYPE_GET_ENVELOPE. In Fortran, the following calls were made:
PARAMETER (LARGE = 1000)
INTEGER TYPE, NI, NA, ND, COMBINER, I(LARGE), D(LARGE), IERROR

INTEGER(KIND=MPI_ADDRESS_KIND) A(LARGE)

! CONSTRUCT DATATYPE TYPE (NOT SHOWN)
CALL MPI_TYPE_GET_ENVELOPE(TYPE, NI, NA, ND, COMBINER, IERROR)

IF ((NI .GT. LARGE) .OR. (NA .GT. LARGE) .OR. (ND .GT. LARGE)) THEN
WRITE (*, *) "NI, NA, OR ND = ", NI, NA, ND, " RETURNED BY MPI_TYPE_GET_ENVELOPE IS LARGER THAN LARGE = ", LARGE
CALL MPI_ABORT(MPI_COMM_WORLD, 99)
ENDIF

CALL MPI_TYPE_GET_CONTENTS(TYPE, NI, NA, ND, I, A, D, IERROR)

or in C the analogous calls of:

#define LARGE 1000
int ni, na, nd, combiner, i[LARGE];
MPI_Aint a[LARGE];
/* construct datatype type (not shown) */
MPI_Type_get_envelope(type, &ni, &na, &nd, &combiner);
if ((ni > LARGE) || (na > LARGE) || (nd > LARGE)) {
fprintf(stderr, "ni, na, or nd = %d %d %d returned by ", ni, na, nd);
fprintf(stderr, "MPI_Type_get_envelope is larger than LARGE = %d
",
LARGE);
MPI_Abort(MPI_COMM_WORLD, 99);
};

MPI_Type_get_contents(type, ni, na, nd, i, a, d);

The C++ code is in analogy to the C code above with the same values returned.
In the descriptions that follow, the lower case name of arguments is used.
If combiner is MPI_COMBINER_NAMED then it is erroneous to call

MPI_TYPE_GET_CONTENTS.

If combiner is MPI_COMBINER_DUP then

Constructor argument C & C++ location Fortran location
oldtype d[0] D(1)
and ni = 0, na = 0, nd = 1.
If combiner is MPI_COMBINER_CONTIGUOUS then

Constructor argument C & C++ location Fortran location
count i[0] I(1)
oldtype d[0] D(1)
and ni = 1, na = 0, nd = 1.
If combiner is MPI_COMBINER_VECTOR then

Constructor argument C & C++ location Fortran location
count i[0] I(1)
blocklength i[1] I(2)
stride i[2] I(3)
oldtype d[0] D(1)
and \( n_i = 3, n_a = 0, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_HVECTOR\_INTEGER} or \texttt{MPI\_COMBINER\_HVECTOR} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>blocklength</td>
<td>[i[1]] ( I(2) )</td>
<td></td>
</tr>
<tr>
<td>stride</td>
<td>[a[0]] ( A(1) )</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = 2, n_a = 1, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_INDEXED\_INTEGER} or \texttt{MPI\_COMBINER\_INDEXED} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_blocklengths</td>
<td>[i[1]] to [i[0]] ( I(2) ) to ( I(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>[i[i[0]]+1] to [2*i[i[0]]] ( I(I(1)+2) ) to ( I^2(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = 2*count+1, n_a = 0, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_HINDEXED\_INTEGER} or \texttt{MPI\_COMBINER\_HINDEXED} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_blocklengths</td>
<td>[i[1]] to [i[0]] ( I(2) ) to ( I(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>[a[0]] to [a[0]-1]] ( A(1) ) to ( A(I(1)) )</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = count+1, n_a = count, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_INDEXED\_BLOCK} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>blocklength</td>
<td>[i[1]] ( I(2) )</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>[i[2]] to [i[i[0]]+1] ( I(3) ) to ( I(I(1)+2) )</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = count+2, n_a = 0, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_STRUCT\_INTEGER} or \texttt{MPI\_COMBINER\_STRUCT} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_blocklengths</td>
<td>[i[1]] to [i[0]] ( I(2) ) to ( I(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>[a[0]] to [a[0]-1]] ( A(1) ) to ( A(I(1)) )</td>
<td></td>
</tr>
<tr>
<td>array_of_types</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = count+1, n_a = count, n_d = count \).

If combiner is \texttt{MPI\_COMBINER\_SUBARRAY} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndims</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_sizes</td>
<td>[i[1]] to [i[0]] ( I(2) ) to ( I(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_subsizes</td>
<td>[i[i[0]]+1] to [2*i[i[0]]] ( I(I(1)+2) ) to ( I^2(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_starts</td>
<td>[i[2]] to [i[i[0]]+1] ( I(3) ) to ( I^3(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>order</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = 3, n_a = 0, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_HVECTOR\_INTEGER} or \texttt{MPI\_COMBINER\_HVECTOR} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>blocklength</td>
<td>[i[1]] ( I(2) )</td>
<td></td>
</tr>
<tr>
<td>stride</td>
<td>[a[0]] ( A(1) )</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = 2, n_a = 1, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_INDEXED\_INTEGER} or \texttt{MPI\_COMBINER\_INDEXED} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_blocklengths</td>
<td>[i[1]] to [i[0]] ( I(2) ) to ( I(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>[i[2]] to [i[i[0]]+1] ( I(3) ) to ( I(I(1)+2) )</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = 2*count+1, n_a = 0, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_INDEXED\_BLOCK} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>blocklength</td>
<td>[i[1]] ( I(2) )</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>[i[2]] to [i[i[0]]+1] ( I(3) ) to ( I(I(1)+2) )</td>
<td></td>
</tr>
<tr>
<td>oldtype</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = count+2, n_a = 0, n_d = 1 \).

If combiner is \texttt{MPI\_COMBINER\_STRUCT\_INTEGER} or \texttt{MPI\_COMBINER\_STRUCT} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_blocklengths</td>
<td>[i[1]] to [i[0]] ( I(2) ) to ( I(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>[a[0]] to [a[0]-1]] ( A(1) ) to ( A(I(1)) )</td>
<td></td>
</tr>
<tr>
<td>array_of_types</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>

and \( n_i = count+1, n_a = count, n_d = count \).

If combiner is \texttt{MPI\_COMBINER\_SUBARRAY} then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>ndims</td>
<td>[i[0]] ( I(1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_sizes</td>
<td>[i[1]] to [i[0]] ( I(2) ) to ( I(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_subsizes</td>
<td>[i[i[0]]+1] to [2*i[i[0]]] ( I(I(1)+2) ) to ( I^2(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>array_of_starts</td>
<td>[i[2]] to [i[i[0]]+1] ( I(3) ) to ( I^3(I(1)+1) )</td>
<td></td>
</tr>
<tr>
<td>order</td>
<td>[d[0]] ( D(1) )</td>
<td></td>
</tr>
</tbody>
</table>
and ni = 3*ndims+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_DARRAY then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>size i[0]</td>
<td>I(1)</td>
<td></td>
</tr>
<tr>
<td>rank i[1]</td>
<td>I(2)</td>
<td></td>
</tr>
<tr>
<td>ndims i[2]</td>
<td>I(3)</td>
<td></td>
</tr>
<tr>
<td>array_of_gsizes i[3] to i[i[2]+2]</td>
<td>I(4) to I(I(3)+3)</td>
<td></td>
</tr>
<tr>
<td>array_of_dists i[i[2]+3] to i[2*i[2]+2]</td>
<td>I(I(3)+4) to I(2*I(3)+3)</td>
<td></td>
</tr>
<tr>
<td>order i[4*i[2]+3]</td>
<td>I(4*I(3)+4)</td>
<td></td>
</tr>
<tr>
<td>oldtype d[0]</td>
<td>D(1)</td>
<td></td>
</tr>
</tbody>
</table>

and ni = 4*ndims+4, na = 0, nd = 1.

If combiner is MPI_COMBINER_F90_REAL then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>p i[0]</td>
<td>I(1)</td>
<td></td>
</tr>
<tr>
<td>r i[1]</td>
<td>I(2)</td>
<td></td>
</tr>
</tbody>
</table>

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_COMPLEX then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>p i[0]</td>
<td>I(1)</td>
<td></td>
</tr>
<tr>
<td>r i[1]</td>
<td>I(2)</td>
<td></td>
</tr>
</tbody>
</table>

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_INTEGER then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>r i[0]</td>
<td>I(1)</td>
<td></td>
</tr>
</tbody>
</table>

and ni = 1, na = 0, nd = 0.

If combiner is MPI_COMBINER_RESIZED then

<table>
<thead>
<tr>
<th>Constructor argument</th>
<th>C &amp; C++ location</th>
<th>Fortran location</th>
</tr>
</thead>
<tbody>
<tr>
<td>lb a[0]</td>
<td>A(1)</td>
<td></td>
</tr>
<tr>
<td>extent a[1]</td>
<td>A(2)</td>
<td></td>
</tr>
<tr>
<td>oldtype d[0]</td>
<td>D(1)</td>
<td></td>
</tr>
</tbody>
</table>

and ni = 0, na = 2, nd = 1.

4.1.14 Examples

The following examples illustrate the use of derived datatypes.

Example 4.13 Send and receive a section of a 3D array.

```
REAL a(100,100,100), e(9,9,9)
INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C extract the section a(1:17:2, 3:11, 2:10)
```
and store it in \( e(\text{,:,:}) \).

\[
\text{CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)}
\]
\[
\text{CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)}
\]

create datatype for a 1D section
\[
\text{CALL MPI_TYPE_VECTOR(9, 1, 2, MPI_REAL, oneslice, ierr)}
\]

create datatype for a 2D section
\[
\text{CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice, ierr)}
\]

create datatype for the entire section
\[
\text{CALL MPI_TYPE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice, threeslice, ierr)}
\]
\[
\text{CALL MPI_TYPE_COMMIT(threeslice, ierr)}
\]
\[
\text{CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)}
\]

Example 4.14  Copy the (strictly) lower triangular part of a matrix.

\[
\text{REAL a(100,100), b(100,100)}
\]
\[
\text{INTEGER disp(100), blocklen(100), ltype, myrank, ierr}
\]
\[
\text{INTEGER status(MPI_STATUS_SIZE)}
\]

C copy lower triangular part of array a onto lower triangular part of array b
\[
\text{CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)}
\]

C compute start and size of each column
\[
\text{DO i=1, 100}
\]
\[
\text{disp(i) = 100*(i-1) + i}
\]
\[
\text{block(i) = 100-i}
\]
\[
\text{END DO}
\]

C create datatype for lower triangular part
\[
\text{CALL MPI_TYPE_INDEXED(100, block, disp, MPI_REAL, ltype, ierr)}
\]
\[
\text{CALL MPI_TYPE_COMMIT(ltype, ierr)}
\]
\[
\text{CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0, e, 9*9*9, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)}
\]

Example 4.15  Transpose a matrix.

\[
\text{REAL a(100,100), b(100,100)}
\]
INTEGER row, xpose, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)

C transpose matrix a onto b
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)

C create datatype for one row
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)

C create datatype for matrix in row-major order
CALL MPI_TYPE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
CALL MPI_TYPE_COMMIT(xpose, ierr)

C send matrix in row-major order and receive in column major order
CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)

Example 4.16 Another approach to the transpose problem:
REAL a(100,100), b(100,100)
INTEGER disp(2), blocklen(2), type(2), row, row1, sizeofreal
INTEGER myrank, ierr
INTEGER status(MPI_STATUS_SIZE)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C transpose matrix a onto b
CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)

C create datatype for one row
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)

C create datatype for one row, with the extent of one real number
disp(1) = 0
disp(2) = sizeofreal
type(1) = row
type(2) = MPI_UB
blocklen(1) = 1
blocklen(2) = 1
CALL MPI_TYPE_STRUCT(2, blocklen, disp, type, row1, ierr)
CALL MPI_TYPE_COMMIT(row1, ierr)
Example 4.17 We manipulate an array of structures.

```c
struct Partstruct
{
    int class; /* particle class */
    double d[6]; /* particle coordinates */
    char b[7]; /* some additional information */
};
```

```c
struct Partstruct particle[1000];
```

```c
/* build datatype describing structure */
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];
MPI_Aint base;

/* compute displacements of structure components */
MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;
MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
```

```c
/* If compiler does padding in mysterious ways,
the following may be safer */
MPI_Datatype type1[4] = {MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_UB};
int blocklen1[4] = {1, 6, 7, 1};
MPI_Aint disp1[4];

/* compute displacements of structure components */
MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp1+1);
MPI_Address( particle[0].b, disp1+2);
MPI_Address( particle+1, disp1+3);
base = disp1[0];
for (i=0; i < 4; i++) disp1[i] -= base;

/* build datatype describing structure */

MPI_Type_struct( 4, blocklen1, disp1, type1, &Particletype);

/* 4.1: send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);

/* 4.2: send only the entries of class zero particles, preceded by the number of such entries */

MPI_Datatype Zparticles; /* datatype describing all particles with class zero (needs to be recomputed if classes change) */
MPI_Datatype Ztype;
int zdisp[1000], j, k;
int zblock[1000], j, k;
MPI_Aint zzdisp[2];
MPI_Datatype zztype[2]; /* compute displacements of class zero particles */
j = 0;
for(i=0; i < 1000; i++)
    if (particle[i].class==0)
    {
        zdisp[j] = i;
zblock[j] = 1;
j++;
    }

/* create datatype for class zero particles */
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Address(&j, zzdisp);
MPI_Address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);
MPI_Type_commit( &Ztype);
MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);

/* A probably more efficient way of defining Zparticles */
/* consecutive particles with index zero are handled as one block */
j=0;
for (i=0; i < 1000; i++)
  if (particle[i].index==0)
    {
    for (k=i+1; (k < 1000)&&(particle[k].index == 0) ; k++);
zdisp[j] = i;
zblock[j] = k-i;
j++;
i = k;
    }
MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);

/* 4.3: send the first two coordinates of all entries */
MPI_Datatype Allpairs; /* datatype for all pairs of coordinates */
MPI_Aint sizeofentry;
MPI_Type_extent( Particletype, &sizeofentry);
/* sizeofentry can also be computed by subtracting the address
of particle[0] from the address of particle[1] */
MPI_Type_hvector( 1000, 2, sizeofentry, MPI_DOUBLE, &Allpairs);
MPI_Type_commit( &Allpairs);
MPI_Send( particle[0].d, 1, Allpairs, dest, tag, comm);

/* an alternative solution to 4.3 */
MPI_Datatype Onepair; /* datatype for one pair of coordinates, with
the extent of one particle entry */
MPI_Aint disp2[3];
MPI_Datatype type2[3] = {MPI_LB, MPI_DOUBLE, MPI_UB};
int blocklen2[3] = {1, 2, 1};
Example 4.18 The same manipulations as in the previous example, but use absolute addresses in datatypes.

struct Partstruct
{
    int class;
    double d[6];
    char b[7];
};
struct Partstruct particle[1000];

    /* build datatype describing first array entry */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int block[3] = {1, 6, 7};
MPI_Aint disp[3];
MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
MPI_Type_struct( 3, block, disp, &Particletype);
/* Particletype describes first array entry -- using absolute addresses */

    /* 5.1: send the entire array */

MPI_Type_commit( &Particletype);
MPI_Send( MPI_BOTTOM, 1000, Particletype, dest, tag, comm);

    /* 5.2: send the entries of class zero,
preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

MPI_Aint zdisp[1000]
int zblock[1000], i, j, k;
int zzblock[2] = {1,1};
MPI_Datatype zztype[2];
MPI_Aint zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
if (particle[i].index==0)
{
    for (k=i+1; (k < 1000)&&(particle[k].index = 0) ; k++);
zdisp[j] = i;
zblock[j] = k-i;
j++;
i = k;
}

MPI_Type_indexed( j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with class zero, using
their absolute addresses*/

/* prepend particle count */
MPI_Address(&j, zzdisp);zzdisp[1] = MPI_BOTTOM;zztype[0] = MPI_INT;zztype[1] = Zparticles;MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);MPI_Type_commit( &Ztype);

MPI_Send( MPI_BOTTOM, 1, Ztype, dest, tag, comm);

Example 4.19 Handling of unions.

union {
    int ival;
    float fval;
}
u[1000]

int  utype;

/* All entries of u have identical type; variable
utype keeps track of their current type */

MPI_Datatype type[2];
4.1. DERIVED DATA TYPES

```c
int blocklen[2] = {1,1};
MPI_Aint disp[2];
MPI_Datatype mpi_utype[2];
MPI_Aint i,j;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */
MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0; disp[1] = j-i;
type[1] = MPI_UB;
type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);
type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);
for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */
MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

Example 4.20 This example shows how a datatype can be decoded. The routine `printdatatype` prints out the elements of the datatype. Note the use of `MPI_Type_free` for datatypes that are not predefined.

```c
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int printdatatype( MPI_Datatype datatype )
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;
    MPI_Type_get_envelope( datatype,
                           &num_ints, &num_adds, &num_dtypes, &combiner );
    switch (combiner) {
    ```
4.2 Pack and Unpack

Some existing communication libraries provide pack/unpack functions for sending noncontiguous data. In these, the user explicitly packs data into a contiguous buffer before sending...
it, and unpacks it from a contiguous buffer after receiving it. Derived datatypes, which are described in Section 4.1, allow one, in most cases, to avoid explicit packing and unpacking. The user specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer. The pack/unpack routines are provided for compatibility with previous libraries. Also, they provide some functionality that is not otherwise available in MPI. For instance, a message can be received in several parts, where the receive operation done on a later part may depend on the content of a former part. Another use is that outgoing messages may be explicitly buffered in user supplied space, thus overriding the system buffering policy. Finally, the availability of pack and unpack operations facilitates the development of additional communication libraries layered on top of MPI.

4.2. PACK AND UNPACK

MPI_PACK(inbuf, incount, datatype, outbuf, outsize, position, comm)

    IN inbuf    input buffer start (choice)
    IN incount  number of input data items (non-negative integer)
    IN datatype datatype of each input data item (handle)
    OUT outbuf  output buffer start (choice)
    IN outsize  output buffer size, in bytes (non-negative integer)
    INOUT position current position in buffer, in bytes (integer)
    IN comm     communicator for packed message (handle)

    int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype, void *outbuf,
                    int outsize, int *position, MPI_Comm comm)

MPI_PACK(INDATA, INOUT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)

    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR

void MPI::Datatype::Pack(const void* inbuf, int incount, void* outbuf,
                          int outsize, int& position, const MPI::Comm &comm) const

    Packs the message in the send buffer specified by inbuf, incount, datatype into the buffer
space specified by outbuf and outsize. The input buffer can be any communication buffer
allowed in MPI_SEND. The output buffer is a contiguous storage area containing outsize
bytes, starting at the address outbuf (length is counted in bytes, not elements, as if it were
a communication buffer for a message of type MPI_PACKED).

    The input value of position is the first location in the output buffer to be used for
packing. position is incremented by the size of the packed message, and the output value
of position is the first location in the output buffer following the locations occupied by the
packed message. The comm argument is the communicator that will be subsequently used
for sending the packed message.
MPI\_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

- **IN** inbuf: input buffer start (choice)
- **IN** insize: size of input buffer, in bytes (non-negative integer)
- **INOUT** position: current position in bytes (integer)
- **OUT** outbuf: output buffer start (choice)
- **IN** outcount: number of items to be unpacked (integer)
- **IN** datatype: datatype of each output data item (handle)
- **IN** comm: communicator for packed message (handle)

```
int MPI\_Unpack(void* inbuf, int insize, int *position, void *outbuf, int outcount, MPI\_Datatype datatype, MPI\_Comm comm)
```

The function unpacks a message into the receive buffer specified by `outbuf`, `outcount`, `datatype` from the buffer space specified by `inbuf` and `insize`. The output buffer can be any communication buffer allowed in `MPI\_RECV`. The input buffer is a contiguous storage area containing `insize` bytes, starting at address `inbuf`. The input value of `position` is the first location in the input buffer occupied by the packed message. `position` is incremented by the size of the packed message, so that the output value of `position` is the first location in the input buffer after the locations occupied by the message that was unpacked. `comm` is the communicator used to receive the packed message.

**Advice to users.** Note the difference between `MPI\_RECV` and `MPI\_UNPACK`: in `MPI\_RECV`, the `count` argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In `MPI\_UNPACK`, the `count` argument specifies the actual number of items that are unpacked; the “size” of the corresponding message is the increment in `position`. The reason for this change is that the “incoming message size” is not predetermined since the user decides how much to unpack; nor is it easy to determine the “message size” from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined a priori. (End of advice to users.)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or `sscanf` in C, for a similar function.)

Unpacks a message into the receive buffer specified by `outbuf`, `outcount`, `datatype` from the buffer space specified by `inbuf` and `insize`. The output buffer can be any communication buffer allowed in `MPI\_RECV`. The input buffer is a contiguous storage area containing `insize` bytes, starting at address `inbuf`. The input value of `position` is the first location in the input buffer occupied by the packed message. `position` is incremented by the size of the packed message, so that the output value of `position` is the first location in the input buffer after the locations occupied by the message that was unpacked. `comm` is the communicator used to receive the packed message.

**Advice to users.** Note the difference between `MPI\_RECV` and `MPI\_UNPACK`: in `MPI\_RECV`, the `count` argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In `MPI\_UNPACK`, the `count` argument specifies the actual number of items that are unpacked; the “size” of the corresponding message is the increment in `position`. The reason for this change is that the “incoming message size” is not predetermined since the user decides how much to unpack; nor is it easy to determine the “message size” from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined a priori. (End of advice to users.)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or `sscanf` in C, for a similar function.)
Several messages can be successively packed into one **packing unit**. This is effected by several successive related calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the “concatenation” of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_unpack.

A packing unit (or a message created by a regular, “typed” send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a subtring of a packing unit necessarily a packing unit. Thus, one cannot concatenate two packing units and then unpack the result as one packing unit; nor can one unpack a substring of a packing unit as a separate packing unit. Each packing unit, that was created by a related sequence of pack calls, or by a regular send, must be unpacked as a unit, by a sequence of related unpack calls.

**Rationale.** The restriction on “atomic” packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment) (**End of rationale**.)

The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

```
MPI_PACK_SIZE(incount, datatype, comm, size)
```

| IN | incout | count argument to packing call (non-negative integer) |
| IN | datatype | datatype argument to packing call (handle) |
| IN | comm | communicator argument to packing call (handle) |
| OUT | size | upper bound on size of packed message, in bytes (non-negative integer) |

```
int MPI_Pack_size(int incout, MPI_Datatype datatype, MPI_Comm comm, int *size)
```

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
```

```
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

```
int MPI::Datatype::Pack_size(int incout, const MPI::Comm& comm) const
```
A call to MPI\_PACK\_SIZE(incount, datatype, comm, size) returns in size an upper bound on the increment in position that is effected by a call to MPI\_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm).

**Rationale.** The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). *(End of rationale.)*

**Example 4.21** An example using MPI\_PACK.

```c
int position, i, j, a[2];
char buff[1000];

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);

if (myrank == 0) {
    /* SENDER CODE */
    position = 0;
    MPI\_Pack(&i, 1, MPI\_INT, buff, 1000, &position, MPI\_COMM\_WORLD);
    MPI\_Pack(&j, 1, MPI\_INT, buff, 1000, &position, MPI\_COMM\_WORLD);
    MPI\_Send( buff, position, MPI\_PACKED, 1, 0, MPI\_COMM\_WORLD);
}
else /* RECEIVER CODE */
    MPI\_Recv( a, 2, MPI\_INT, 0, 0, MPI\_COMM\_WORLD)
```

**Example 4.22** An elaborate example.

```c
int position, i;
float a[1000];
char buff[1000];

MPI\_Comm\_rank(MPI\_Comm\_world, &myrank);

if (myrank == 0) {
    /* SENDER CODE */
    int len[2];
    MPI\_Aint disp[2];
    MPI\_Datatype type[2], newtype;
    /* build datatype for i followed by a[0]...a[i-1] */

    len[0] = 1;
    len[1] = i;
```
4.2. PACK AND UNPACK

MPI_Address( &i, disp);
MPI_Address( a, disp+1);
type[0] = MPI_INT;
type[1] = MPI_FLOAT;
MPI_Type_struct( 2, len, disp, type, &newtype);
MPI_Type_commit( &newtype);

/* Pack i followed by a[0]...a[i-1]*/
position = 0;
MPI_Pack( MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);

/* Send */
MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);

/* *****/
One can replace the last three lines with
MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);

} else if (myrank == 1){

/* RECEIVER CODE */
MPI_Status status;

/* Receive */
MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);

/* Unpack i */
position = 0;
MPI_Unpack( buff, 1000, &position, &i, MPI_INT, MPI_COMM_WORLD);

/* Unpack a[0]...a[i-1]*/
MPI_Unpack( buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);

}

Example 4.23
Each process sends a count, followed by count characters to the root; the
root concatenates all characters into one string.

int count, gsize, counts[64], totalcount, k1, k2, k,
disps[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;

Example 4.23
Each process sends a count, followed by count characters to the root; the
root concatenates all characters into one string.

int count, gsize, counts[64], totalcount, k1, k2, k,
disps[64], position, concat_pos;
char chr[100], *lbuf, *rbuf, *cbuf;
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

/* allocate local pack buffer */
MPI_Pack_size(1, MPI_INT, comm, &k1);
MPI_Pack_size(count, MPI_CHAR, comm, &k2);
k = k1+k2;
lbuf = (char *)malloc(k);

/* pack count, followed by count characters */
position = 0;MPI_Pack(&count, 1, MPI_INT, lbuf, k, &position, comm);
MPI_Pack(chr, count, MPI_CHAR, lbuf, k, &position, comm);
if (myrank != root) {
    /* gather at root sizes of all packed messages */
    MPI_Gather( &position, 1, MPI_INT, NULL, NULL, NULL, root, comm);
    /* gather at root packed messages */
    MPI_Gatherv( &buf, position, MPI_PACKED, NULL, NULL, NULL, NULL, root, comm);
} else { /* root code */
    /* gather sizes of all packed messages */
    MPI_Gather( &position, 1, MPI_INT, counts, 1, MPI_INT, root, comm);
    /* gather all packed messages */
    displs[0] = 0;
    for (i=1; i < gsize; i++)
        displs[i] = displs[i-1] + counts[i-1];
    totalcount = displs[gsize-1] + counts[gsize-1];
    rbuf = (char *)malloc(totalcount);
    cbuf = (char *)malloc(totalcount);
    MPI_Gatherv( lbuf, position, MPI_PACKED, rbuf, 
                 counts, displs, MPI_PACKED, root, comm);
    /* unpack all messages and concatenate strings */
    concat_pos = 0;
    for (i=0; i < gsize; i++) {
        position = 0;
        MPI_Unpack( rbuf+displs[i], totalcount-displs[i], 
                   &position, &count, 1, MPI_INT, comm);
        MPI_Unpack( rbuf+displs[i], totalcount-displs[i], 
                   &position, cbuf+concat_pos, count, MPI_CHAR, comm);
        concat_pos += count;
    }
}
4.3 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the “external32” data format specified in Section 13.5.2, and calculate the size needed for packing. Their first arguments specify the data format, for future extensibility, but currently the only valid value of the datarep argument is “external32.”

Advice to users. These functions could be used, for example, to send typed data in a portable format from one MPI implementation to another. (End of advice to users.)

The buffer will contain exactly the packed data, without headers. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message and further specifies the exact format of the data. Since MPI_PACK may (and is allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed with MPI_PACK_EXTERNAL. (End of rationale.)

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)

IN datarep data representation (string)
IN inbuf input buffer start (choice)
IN incount number of input data items (integer)
IN datatype datatype of each input data item (handle)
OUT outbuf output buffer start (choice)
IN outsize output buffer size, in bytes (integer)
INOUT position current position in buffer, in bytes (integer)

int MPI_Pack_external(char *datarep, void *inbuf, int incount,
MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
MPI_Aint position)

MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
POSITION, IERROR)

INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
CHARACTER(*) DATAREP
<TYPE> INBUF(*), OUTBUF(*)

void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
int incount, void* outbuf, MPI::Aint outsize,
MPI::Aint& position) const

4.3 Canonical MPI_PACK and MPI_UNPACK

These functions read/write data to/from the buffer in the “external32” data format specified in Section 13.5.2, and calculate the size needed for packing. Their first arguments specify the data format, for future extensibility, but currently the only valid value of the datarep argument is “external32.”

Advice to users. These functions could be used, for example, to send typed data in a portable format from one MPI implementation to another. (End of advice to users.)

The buffer will contain exactly the packed data, without headers. MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

Rationale. MPI_PACK_EXTERNAL specifies that there is no header on the message and further specifies the exact format of the data. Since MPI_PACK may (and is allowed to) use a header, the datatype MPI_PACKED cannot be used for data packed with MPI_PACK_EXTERNAL. (End of rationale.)

MPI_PACK_EXTERNAL(datarep, inbuf, incount, datatype, outbuf, outsize, position)

IN datarep data representation (string)
IN inbuf input buffer start (choice)
IN incount number of input data items (integer)
IN datatype datatype of each input data item (handle)
OUT outbuf output buffer start (choice)
IN outsize output buffer size, in bytes (integer)
INOUT position current position in buffer, in bytes (integer)

int MPI_Pack_external(char *datarep, void *inbuf, int incount,
MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
MPI_Aint position)

MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
POSITION, IERROR)

INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION
INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
CHARACTER(*) DATAREP
<TYPE> INBUF(*), OUTBUF(*)

void MPI::Datatype::Pack_external(const char* datarep, const void* inbuf,
int incount, void* outbuf, MPI::Aint outsize,
MPI::Aint& position) const
CHAPTER 4. DATA TYPES

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, position)

IN datarep   data representation (string)
IN inbuf     input buffer start (choice)
IN insize    input buffer size, in bytes (integer)
INOUT position current position in buffer, in bytes (integer)
OUT outbuf   output buffer start (choice)
IN outcount  number of output data items (integer)
IN datatype  datatype of output data item (handle)

int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
                        MPI_Aint *position, void *outbuf, int outcount,
                        MPI_Datatype datatype)

MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTSIZE,
DATATYPE, IERROR)
INTEGER OUTCOUNT, DATATYPE, IERRORINTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION

void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf,
                                     MPI::Aint insize, MPI::Aint& position, void* outbuf,
                                     int outcount) const

MPI_PACK_EXTERNAL_SIZE( datarep, incount, datatype, size )

IN datarep   data representation (string)
IN incount   number of input data items (integer)
IN datatype  datatype of each input data item (handle)
OUT size     output buffer size, in bytes (integer)

int MPI_Pack_external_size(char *datarep, int incount,
                           MPI_Datatype datatype, MPI_Aint *size)

MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, SIZE, IERROR

void MPI::Datatype::Pack_external_size(const char* datarep,
                                       int incount) const

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, position)

IN datarep   data representation (string)
IN inbuf     input buffer start (choice)
IN insize    input buffer size, in bytes (integer)
INOUT position current position in buffer, in bytes (integer)
OUT outbuf   output buffer start (choice)
IN outcount  number of output data items (integer)
IN datatype  datatype of output data item (handle)

int MPI_Unpack_external(char *datarep, void *inbuf, MPI_Aint insize,
                        MPI_Aint *position, void *outbuf, int outcount,
                        MPI_Datatype datatype)

MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT,
DATATYPE, IERROR)
INTEGER OUTCOUNT, DATATYPE, IERRORINTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION

void MPI::Datatype::Unpack_external(const char* datarep, const void* inbuf,
                                     MPI::Aint insize, MPI::Aint& position, void* outbuf,
                                     int outcount) const

MPI_PACK_EXTERNAL_SIZE( datarep, incount, datatype, size )

IN datarep   data representation (string)
IN incount   number of input data items (integer)
IN datatype  datatype of each input data item (handle)
OUT size     output buffer size, in bytes (integer)

int MPI_Pack_external_size(char *datarep, int incount,
                           MPI_Datatype datatype, MPI_Aint *size)

MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, SIZE, IERROR

void MPI::Datatype::Pack_external_size(const char* datarep,
                                       int incount) const
Chapter 5

Collective Communication

5.1 Introduction and Overview

Collective communication is defined as communication that involves a group or groups of processes. The functions of this type provided by MPI are the following:

- **MPI_BARRIER**: Barrier synchronization across all members of a group (Section 5.3).
- **MPI_BCAST**: Broadcast from one member to all members of a group (Section 5.4).
  This is shown as “broadcast” in Figure 5.1.
- **MPI_GATHER, MPI_GATHERV**: Gather data from all members of a group to one member (Section 5.5).
  This is shown as “gather” in Figure 5.1.
- **MPI_SCATTER, MPI_SCATTERV**: Scatter data from one member to all members of a group (Section 5.6).
  This is shown as “scatter” in Figure 5.1.
- **MPI_ALLGATHER, MPI_ALLGATHERV**: A variation on Gather where all members of a group receive the result (Section 5.7).
  This is shown as “allgather” in Figure 5.1.
- **MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW**: Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all) (Section 5.8).
  This is shown as “alltoall” in Figure 5.1.
- **MPI_ALLREDUCE, MPI_REDUCE**: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group and a variation where the result is returned to only one member (Section 5.9).
- **MPI_REDUCE_SCATTER**: A combined reduction and scatter operation (Section 5.10).
- **MPI_SCAN, MPI_EXSCAN**: Scan across all members of a group (also called prefix) (Section 5.11).

One of the key arguments in a call to a collective routine is a communicator that defines the group or groups of participating processes and provides a context for the operation. This is discussed further in Section 5.2. The syntax and semantics of the collective operations are defined to be consistent with the syntax and semantics of the point-to-point operations. Thus, general datatypes are allowed and must match between sending and receiving processes as specified in Chapter 4. Several collective routines such as broadcast...
Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data $A_0$, but after the broadcast all processes contain it.
and gather have a single originating or receiving process. Such a process is called the root. Some arguments in the collective functions are specified as "significant only at root," and are ignored for all participants except the root. The reader is referred to Chapter 4 for information concerning communication buffers, general datatypes and type matching rules, and to Chapter 6 for information on how to define groups and create communicators.

The type-matching conditions for the collective operations are more strict than the corresponding conditions between sender and receiver in point-to-point. Namely, for collective operations, the amount of data sent must exactly match the amount of data specified by the receiver. Different type maps (the layout in memory, see Section 4.1) between sender and receiver are still allowed.

Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise implied by in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. A more detailed discussion of correct use of collective routines is found in Section 5.12.

Rationale. The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

The collective operations do not accept a message tag argument. If future revisions of MPI define non-blocking collective functions, then tags (or a similar mechanism) might need to be added so as to allow the dis-ambiguation of multiple, pending, collective operations. (End of rationale.)

Advice to users. It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.

On the other hand, a correct, portable program must allow for the fact that a collective call may be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it. These issues are discussed further in Section 5.12. (End of advice to users.)

Advice to implementors. While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden,
special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.12. (End of advice to implementors.)

Many of the descriptions of the collective routines provide illustrations in terms of blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are not correct MPI programs; for purposes of simplicity, they often assume infinite buffering.

5.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter 6. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: intra-communicators and inter-communicators. An intracommunicator can be thought of as an identifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context.

5.2.1 Specifics for Intracommunicator Collective Operations

All processes in the group identified by the intracommunicator must call the collective routine with matching arguments.

In many cases, collective communication can occur “in place” for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

Rationale. The “in place” operations are provided to reduce unnecessary memory motion by both the MPI implementation and by the user. Note that while the simple check of testing whether the send and receive buffers have the same address will work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits aliasing of arguments; the approach of using a special value to denote “in place” operation eliminates that difficulty. (End of rationale.)

Advice to users. By allowing the “in place” option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes INTENT must mark these as INOUT, not OUT.

Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its use that MPI_BOTTOM has.

Some intracommunicator collective operations do not support the “in place” option (e.g., MPI_ALLTOALLV). (End of advice to users.)

special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.12. (End of advice to implementors.)

Many of the descriptions of the collective routines provide illustrations in terms of blocking MPI point-to-point routines. These are intended solely to indicate what data is sent or received by what process. Many of these examples are not correct MPI programs; for purposes of simplicity, they often assume infinite buffering.

5.2 Communicator Argument

The key concept of the collective functions is to have a group or groups of participating processes. The routines do not have group identifiers as explicit arguments. Instead, there is a communicator argument. Groups and communicators are discussed in full detail in Chapter 6. For the purposes of this chapter, it is sufficient to know that there are two types of communicators: intra-communicators and inter-communicators. An intracommunicator can be thought of as an identifier for a single group of processes linked with a context. An intercommunicator identifies two distinct groups of processes linked with a context.

5.2.1 Specifics for Intracommunicator Collective Operations

All processes in the group identified by the intracommunicator must call the collective routine with matching arguments.

In many cases, collective communication can occur “in place” for intracommunicators, with the output buffer being identical to the input buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send buffer or the receive buffer argument, depending on the operation performed.

Rationale. The “in place” operations are provided to reduce unnecessary memory motion by both the MPI implementation and by the user. Note that while the simple check of testing whether the send and receive buffers have the same address will work for some cases (e.g., MPI_ALLREDUCE), they are inadequate in others (e.g., MPI_GATHER, with root not equal to zero). Further, Fortran explicitly prohibits aliasing of arguments; the approach of using a special value to denote “in place” operation eliminates that difficulty. (End of rationale.)

Advice to users. By allowing the “in place” option, the receive buffer in many of the collective calls becomes a send-and-receive buffer. For this reason, a Fortran binding that includes INTENT must mark these as INOUT, not OUT.

Note that MPI_IN_PLACE is a special kind of value; it has the same restrictions on its use that MPI_BOTTOM has.

Some intracommunicator collective operations do not support the “in place” option (e.g., MPI_ALLTOALLV). (End of advice to users.)
5.2. COMMUNICATOR ARGUMENT

5.2.2 Applying Collective Operations to Intercommunicators

To understand how collective operations apply to intercommunicators, we can view most MPI intracommunicator collective operations as fitting one of the following categories (see, for instance, [43]):

All-To-All All processes contribute to the result. All processes receive the result.
- MPI_ALLGATHER, MPI_ALLGATHERV
- MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW
- MPI_ALLREDUCE, MPI_ALLREDUCE_SCATTER

All-To-One All processes contribute to the result. One process receives the result.
- MPI_GATHER, MPI_GATHERV
- MPI_REDUCE

One-To-All One process contributes to the result. All processes receive the result.
- MPI_BCAST
- MPI_SCATTER, MPI_SCATTERV

Other Collective operations that do not fit into one of the above categories.
- MPI_SCAN, MPI_EXSCAN
- MPI_BARRIER

The MPI_BARRIER operation does not fit into this classification since no data is being moved (other than the implicit fact that a barrier has been called). The data movement patterns of MPI_SCAN and MPI_EXSCAN do not fit this taxonomy.

The application of collective communication to intercommunicators is best described in terms of two groups. For example, an all-to-all MPI_ALLGATHER operation can be described as collecting data from all members of one group with the result appearing in all members of the other group (see Figure 5.2). As another example, a one-to-all MPI_BCAST operation sends data from one member of one group to all members of the other group. Collective computation operations such as MPI_REDUCE_SCATTER have a similar interpretation (see Figure 5.3). For intracommunicators, these two groups are the same. For intercommunicators, these two groups are distinct. For the all-to-all operations, each such operation is described in two phases, so that it has a symmetric, full-duplex behavior.

The following collective operations also apply to intercommunicators:
- MPI_BARRIER,
- MPI_BCAST,
- MPI_GATHER, MPI_GATHERV,
- MPI_SCATTER, MPI_SCATTERV,
- MPI_ALLGATHER, MPI_ALLGATHERV,
• MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLTOALLW,
• MPI_ALLREDUCE, MPI_REDUCE,
• MPI_REDUCE_SCATTER.

In C++, the bindings for these functions are in the MPI::Comm class. However, since the collective operations do not make sense on a C++ MPI::Comm (as it is neither an intercommunicator nor an intracommunicator), the functions are all pure virtual.

5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective routine. In addition, processes in the same group must call the routine with matching arguments.

Note that the “in place” option for intracommunicators does not apply to intercommunicators since in the intercommunicator case there is no communication from a process to itself.

For intercommunicator collective communication, if the operation is rooted (e.g., broadcast, gather, scatter), then the transfer is unidirectional. The direction of the transfer is indicated by a special value of the root argument. In this case, for the group containing the root process, all processes in the group must call the routine using a special argument for the root. For this, the root process uses the special root value MPI_ROOT; all other processes in the same group as the root use MPI_PROC_NULL. All processes in the other group (the group that is the remote group relative to the root process) must call the collective routine and provide the rank of the root. If the operation is unrooted (e.g., alltoall), then the transfer is bidirectional.

Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.
Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

**Rationale.** Rooted operations are unidirectional by nature, and there is a clear way of specifying direction. Non-rooted operations, such as all-to-all, will often occur as part of an exchange, where it makes sense to communicate in both directions at once. *(End of rationale.)*

### 5.3 Barrier Synchronization

```c
MPI_BARRIER( comm )

IN comm communicator (handle)
```

```c
int MPI_Barrier(MPI_Comm comm )

MPI_BARRIER(MPI_COMM, IERROR)

INTEGER COMM, IERROR
```

```cpp
void MPI::Comm::Barrier() const = 0
```

If `comm` is an intracommunicator, `MPI_BARRIER` blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

If `comm` is an intercommunicator, the barrier is performed across all processes in the intercommunicator. In this case, all processes in one group (group A) of the intercommunicator may exit the barrier when all of the processes in the other group (group B) have entered the barrier.
5.4 Broadcasting

MPI_BCAST( buffer, count, datatype, root, comm )

INOUT buffer starting address of buffer (choice)
IN count number of entries in buffer (non-negative integer)
IN datatype data type of buffer (handle)
IN root rank of broadcast root (integer)
IN comm communicator (handle)

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

<int> BUFFER(*)INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

void MPI::Comm::Bcast(void* buffer, int count, const MPI::Datatype& datatype, int root) const = 0

void MPI::Comm::Bcast(void* buffer, int count, int root)

If comm is an intracommunicator, MPI_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group using the same arguments for comm and root. On return, the content of root’s buffer is copied to all other processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

The “in place” option is not meaningful here.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

Example 5.1 Broadcast 100 ints from process 0 to every process in the group.

MPI_Comm comm;
int array[100];
As in many of our example code fragments, we assume that some of the variables (such as \texttt{comm} in the above) have been assigned appropriate values.

5.5 Gather

\begin{verbatim}
MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
\end{verbatim}

\begin{tabular}{ll}
IN & sendbuf \hspace{1cm} \text{starting address of send buffer (choice)}
\hline
IN & sendcount \hspace{1cm} \text{number of elements in send buffer (non-negative integer)}
\hline
IN & sendtype \hspace{1cm} \text{data type of send buffer elements (handle)}
\hline
OUT & recvbuf \hspace{1cm} \text{address of receive buffer (choice, significant only at root)}
\hline
IN & recvcount \hspace{1cm} \text{number of elements for any single receive (non-negative integer, significant only at root)}
\hline
IN & recvtype \hspace{1cm} \text{data type of recv buffer elements (significant only at root)}
\hline
IN & root \hspace{1cm} \text{rank of receiving process (integer)}
\hline
IN & comm \hspace{1cm} \text{communicator (handle)}
\end{tabular}

int \texttt{MPI\_Gather(void* sendbuf, int sendcount, MPI\_Datatype sendtype,}
void* recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPIC\_Comm\_comm)}

\begin{verbatim}
MPI\_GATHER\_sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
\end{verbatim}

\begin{verbatim}
void MPI::Comm::Gather(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype,
void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int root) const = 0
\end{verbatim}

If \texttt{comm} is an intracommunicator, each process (root process included) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order. The outcome is as if each of the \( n \) processes in the group (including the root process) had executed a call to

\begin{verbatim}
MPI\_Send(sendbuf, sendcount, sendtype, root, ...),
\end{verbatim}

and the root had executed \( n \) calls to

\begin{verbatim}
MPI\_Recv(recvbuf + i \cdot recvcount \cdot extent(recvtype), recvcount, recvtype, i, ...),
\end{verbatim}
where \( \text{extent(recvtype)} \) is the type extent obtained from a call to \( \text{MPI\_Type\_extent()} \).

An alternative description is that the \( n \) messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to \( \text{MPI\_RECV(recvbuf, recvcount, n, recvtype, ...)} \).

The receive buffer is ignored for all non-root processes. General, derived datatypes are allowed for both \( \text{sendtype} \) and \( \text{recvtype} \). The type signature of \( \text{sendcount} \), \( \text{sendtype} \) on each process must be equal to the type signature of \( \text{recvcount} \), \( \text{recvtype} \) at the root. This implies that the amount of data sent must equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments \( \text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{root} \), and \( \text{comm} \) are significant. The arguments \( \text{root} \) and \( \text{comm} \) must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the \( \text{recvcount} \) argument at the root indicates the number of items it receives from each process, not the total number of items it receives.

The “in place” option for intracommunicators is specified by passing \( \text{MPI\_IN\_PLACE} \) as the value of \( \text{sendbuf} \) at the root. In such a case, \( \text{sendcount} \) and \( \text{sendtype} \) are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

If \( \text{comm} \) is an intercommunicator, then the call involves all processes in the intercommunicator, with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument \( \text{root} \), which is the rank of the root in group A. The root passes the value \( \text{MPI\_ROOT} \) in \( \text{root} \). All other processes in group A pass the value \( \text{MPI\_PROC\_NULL} \) in \( \text{root} \). Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

```
MPI\_GATHERV( sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
```

<table>
<thead>
<tr>
<th>IN</th>
<th>sendbuf</th>
<th>starting address of send buffer (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendcount</td>
<td>number of elements in send buffer (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>sendtype</td>
<td>data type of send buffer elements (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
<td>address of receive buffer (choice, significant only at root)</td>
</tr>
</tbody>
</table>

where \( \text{extent(recvtype)} \) is the type extent obtained from a call to \( \text{MPI\_Type\_extent()} \).

An alternative description is that the \( n \) messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to \( \text{MPI\_RECV(recvbuf, recvcount, n, recvtype, ...)} \).

The receive buffer is ignored for all non-root processes. General, derived datatypes are allowed for both \( \text{sendtype} \) and \( \text{recvtype} \). The type signature of \( \text{sendcount} \), \( \text{sendtype} \) on each process must be equal to the type signature of \( \text{recvcount} \), \( \text{recvtype} \) at the root. This implies that the amount of data sent must equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

All arguments to the function are significant on process root, while on other processes, only arguments \( \text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{root} \) and \( \text{comm} \) are significant. The arguments \( \text{root} \) and \( \text{comm} \) must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the \( \text{recvcount} \) argument at the root indicates the number of items it receives from each process, not the total number of items it receives.

The “in place” option for intracommunicators is specified by passing \( \text{MPI\_IN\_PLACE} \) as the value of \( \text{sendbuf} \) at the root. In such a case, \( \text{sendcount} \) and \( \text{sendtype} \) are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

If \( \text{comm} \) is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument \( \text{root} \), which is the rank of the root in group A. The root passes the value \( \text{MPI\_ROOT} \) in \( \text{root} \). All other processes in group A pass the value \( \text{MPI\_PROC\_NULL} \) in \( \text{root} \). Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

```
MPI\_GATHERV( sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
```

<table>
<thead>
<tr>
<th>IN</th>
<th>sendbuf</th>
<th>starting address of send buffer (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>sendcount</td>
<td>number of elements in send buffer (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>sendtype</td>
<td>data type of send buffer elements (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>recvbuf</td>
<td>address of receive buffer (choice, significant only at root)</td>
</tr>
</tbody>
</table>
5.5. **GATHER**

<table>
<thead>
<tr>
<th>IN recvcounts</th>
<th>non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN displs</td>
<td>integer array (of length group size). Entry 1 specifies the displacement relative to recvbuf at which to place the incoming data from process 1 (significant only at root)</td>
</tr>
<tr>
<td>IN recvtype</td>
<td>data type of recv buffer elements (significant only at root) (handle)</td>
</tr>
<tr>
<td>IN root</td>
<td>rank of receiving process (integer)</td>
</tr>
<tr>
<td>IN comm</td>
<td>communicator (handle)</td>
</tr>
</tbody>
</table>

```c
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, 
                 void* recvbuf, int *recvcounts, int *displs,
                 MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDDTYPE,_RECVBUF, _RECVCOUNTS, DISPLS, 
             _RECVTYPE, ROOT, COMM, IERROR)

void MPI::Comm::Gatherv(const void* sendbuf, int sendcount, const 
                        MPI::Datatype& sendtype, void* recvbuf, 
                        const int recvcounts[], const int displs[], 
                        const MPI::Datatype& recvtype, const int root) const = 0
```

If `comm` is an intracommunicator, the outcome is as if each process, including the root process, sends a message to the root,

```c
MPI_Send(sendbuf, sendcount, sendtype, root, ...),
```
and the root executes `n` receives,

```c
MPI_Recv(recvbuf + displs[j] * extent(recvtype), recvcounts[j], recvtype, i, ...).
```

The data received from process `j` is placed into `recvbuf` of the root process beginning at offset `displs[j]` elements (in terms of the `recvtype`).

The receive buffer is ignored for all non-root processes.

The type signature implied by `sendcount`, `sendtype` on process `i` must be equal to the type signature implied by `recvcounts[j]`, `recvtype` at the root. This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed, as illustrated in Example 5.6.

All arguments to the function are significant on process `root`, while on other processes, only arguments `sendbuf`, `sendcount`, `sendtype`, `root`, and `comm` are significant. The arguments `root` and `comm` must have identical values on all processes.
The specification of counts, types, and displacements should not cause any location on the root to be written more than once. Such a call is erroneous.

The “in place” option for intracommunicators is specified by passing `MPI_IN_PLACE` as the value of `sendbuf` at the root. In such a case, `sendcount` and `sendtype` are ignored, and the contribution of the root to the gathered vector is assumed to be already in the correct place in the receive buffer.

If `comm` is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument `root`, which is the rank of the root in group A. The root passes the value `MPI_ROOT` in `root`. All other processes in group A pass the value `MPI_PROC_NULL` in `root`. Data is gathered from all processes in group B to the root. The send buffer arguments of the processes in group B must be consistent with the receive buffer argument of the root.

5.5.1 Examples using MPI_GATHER, MPI_GATHERV

The examples in this section use intracommunicators.

**Example 5.2** Gather 100 ints from every process in group to root. See figure 5.4.

```c
MPI_Comm comm; int gsize, sendarray[100]; int root, *rbuf;...
MPI_Comm_size(comm, &gsize); rbuf = (int *)malloc(gsize*100*sizeof(int)); MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

**Example 5.3** Previous example modified – only the root allocates memory for the receive buffer.

```c
MPI_Comm comm; int gsize, sendarray[100]; int root, myrank, *rbuf;...
MPI_Comm_rank(comm, &myrank); if (myrank == root) {
    MPI_Comm_size(comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

**Example 5.4** Do the same as the previous example, but use a derived datatype. Note that the type cannot be the entire set of `gsize*100` ints since type matching is defined pairwise between the root and each process in the gather.

```c
MPI_Comm comm; int gsize, sendarray[100];
```
5.5. **GATHER**

**Figure 5.4:** The root process gathers 100 ints from each process in the group.

```c
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

**Example 5.5** Now have each process send 100 ints to root, but place each set (of 100) stride ints apart at receiving end. Use `MPI_GATHERV` and the displs argument to achieve this effect. Assume stride ≥ 100. See Figure 5.5.

```c
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;
...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
  displs[i] = i*stride;
  rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT, root, comm);
```

Note that the program is erroneous if stride < 100.

**Example 5.6** Same as Example 5.5 on the receiving side, but send the 100 ints from the 0th column of a 100×150 int array, in C. See Figure 5.6.

```c
MPI_Comm comm;
```

---

**Figure 5.6:** The root process gathers 100 ints from each process in the group.

```c
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

**Example 5.5** Now have each process send 100 ints to root, but place each set (of 100) stride ints apart at receiving end. Use `MPI_GATHERV` and the displs argument to achieve this effect. Assume stride ≥ 100. See Figure 5.5.

```c
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;
...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
  displs[i] = i*stride;
  rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT, root, comm);
```

Note that the program is erroneous if stride < 100.

**Example 5.6** Same as Example 5.5 on the receiving side, but send the 100 ints from the 0th column of a 100×150 int array, in C. See Figure 5.6.

```c
MPI_Comm comm;
```
Figure 5.5: The root process gathers 100 ints from each process in the group, each set is placed \textit{stride} ints apart.

Figure 5.6: The root process gathers column 0 of a 100\times150 C array, and each set is placed \textit{stride} ints apart.

Example 5.7 Process \textit{i} sends \((100-i)\) ints from the \textit{i}-th column of a 100 \times 150 \textit{int}
array, in C. It is received into a buffer with stride, as in the previous two examples. See Figure 5.7.

```c
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts;
...
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100-i; /* note change from previous example */
} /* Create datatype for the column we are sending */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );/* sptr is the address of start of “myrank” column */
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT,
            root, comm);
Note that a different amount of data is received from each process.
```

**Example 5.8** Same as Example 5.7, but done in a different way at the sending end. We create a datatype that causes the correct striding at the sending end so that we read a column of a C array. A similar thing was done in Example 4.16, Section 4.1.14.
Example 5.9 Same as Example 5.7 at sending side, but at receiving side we make the stride between received blocks vary from block to block. See Figure 5.8.
5.5. GATHER

Figure 5.8: The root process gathers $100-i$ ints from column $i$ of a $100 \times 150$ C array, and each set is placed $\text{stride}[i]$ ints apart (a varying stride).

```c
/* set up displs and rcounts vectors first */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
/* the required buffer size for rbuf is now easily obtained */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));

/* Create datatype for the column we are sending */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 1, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```

Example 5.10 Process $i$ sends $\text{num}$ ints from the $i$-th column of a $100 \times 150$ int array, in C. The complicating factor is that the various values of $\text{num}$ are not known to root, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```c
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,types[2];
int *displs,1,*rcounts,num;
...```

Example 5.10 Process $i$ sends $\text{num}$ ints from the $i$-th column of a $100 \times 150$ int array, in C. The complicating factor is that the various values of $\text{num}$ are not known to root, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```c
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,types[2];
int *displs,1,*rcounts,num;
...```
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
/* First, gather nums to root */
rcounts = (int *)malloc(gsize*sizeof(int));
MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
/* root now has correct rcounts, using these we set displs[] so */
/* that data is placed contiguously (or concatenated) at receive end */
displs = (int *)malloc(gsize*sizeof(int));
displs[0] = 0;
for (i=1; i<gsize; ++i) {
    displs[i] = displs[i-1]+rcounts[i-1];
}
/* And, create receive buffer */
rbuf = (int *)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])
        *sizeof(int));
/* Create datatype for one int, with extent of entire row */
disp[0] = 0; disp[1] = 150*sizeof(int);
type[0] = MPI_INT; type[1] = MPI_UB;
blocklen[0] = 1; blocklen[1] = 1;
MPI_Type_struct( 2, blocklen, disp, type, &stype );
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, num, stype, rbuf, rcounts, displs, MPI_INT,
              root, comm);
## 5.6 Scatter

### MPI_SCATTER

**IN**
- `sendbuf`: address of send buffer (choice, significant only at root)
- `sendcount`: number of elements sent to each process (non-negative integer, significant only at root)
- `sendtype`: data type of send buffer elements (significant only at root)

**OUT**
- `recvbuf`: address of receive buffer (choice)
- `recvcount`: number of elements in receive buffer (non-negative integer)
- `recvtype`: data type of receive buffer elements (handle)
- `root`: rank of sending process (integer)
- `comm`: communicator (handle)

**int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

**MPI_SCATTER**

- `SENDBUF`, `SENDCOUNT`, `SENDTYPE`, `RECVBUF`, `RECVCOUNT`, `RECVTYPE`, `ROOT`, `COMM`, `ERROR`
- `<type> SENDBUF(*)`, `RECVBUF(*)`
- `INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR`

**void MPI::Comm::Scatter(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recvbuf, int recvcount, const MPI::Datatype& recvtype, int root) const = 0

**MPI_SCATTER** is the inverse operation to **MPI_GATHER**.

If **comm** is an intracommunicator, the outcome is as if the root executed `n` send operations,

- **MPI_Send(sendbuf + i * sendcount * extent(sendtype), sendcount, sendtype, i, ...),** and each process executed a receive,

- **MPI_Recv(recvbuf, recvcount, recvtype, i,...)**.

An alternative description is that the root sends a message with **MPI_Send(sendbuf, sendcount, n, sendtype, ...).** This message is split into `n` equal segments, the `i`-th segment is sent to the `i`-th process in the group, and each process receives this message as above.

The send buffer is ignored for all non-root processes.

The type signature associated with `sendcount`, `sendtype` at the root must be equal to the type signature associated with `recvcount`, `recvtype` at all processes (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.
All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

Rationale. Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (End of rationale.)

The “in place” option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root “sends” no data to itself. The scattered vector is still assumed to contain \( n \) segments, where \( n \) is the group size; the \( i \)-th segment, which root should “send to itself,” is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

```c
MPI_SCATTERV( sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm )
```

| IN   | sendbuf           | address of send buffer (choice, significant only at root) |
| IN   | sendcounts        | non-negative integer array (of length group size) specifying the number of elements to send to each processor |
| IN   | displs            | integer array (of length group size). Entry \( i \) specifies the displacement (relative to sendbuf from which to take the outgoing data to process \( i \)) |
| IN   | sendtype          | data type of send buffer elements (handle) |
| OUT  | recvbuf           | address of receive buffer (choice) |
| IN   | recvcount         | number of elements in receive buffer (non-negative integer) |
| IN   | recvtype          | data type of receive buffer elements (handle) |
| IN   | root              | rank of sending process (integer) |
| IN   | comm              | communicator (handle) |

```c
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```c
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPER, ROOT, COMM, IERROR)
```

All arguments to the function are significant on process root, while on other processes, only arguments recvbuf, recvcount, recvtype, root, and comm are significant. The arguments root and comm must have identical values on all processes.

The specification of counts and types should not cause any location on the root to be read more than once.

Rationale. Though not needed, the last restriction is imposed so as to achieve symmetry with MPI_GATHER, where the corresponding restriction (a multiple-write restriction) is necessary. (End of rationale.)

The “in place” option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such case, recvcount and recvtype are ignored, and root “sends” no data to itself. The scattered vector is still assumed to contain \( n \) segments, where \( n \) is the group size; the \( i \)-th segment, which root should “send to itself,” is not moved.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is scattered from the root to all processes in group B. The receive buffer arguments of the processes in group B must be consistent with the send buffer argument of the root.

```c
MPI_SCATTERV( sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm )
```

| IN   | sendbuf           | address of send buffer (choice, significant only at root) |
| IN   | sendcounts        | non-negative integer array (of length group size) specifying the number of elements to send to each processor |
| IN   | displs            | integer array (of length group size). Entry \( i \) specifies the displacement (relative to sendbuf from which to take the outgoing data to process \( i \)) |
| IN   | sendtype          | data type of send buffer elements (handle) |
| OUT  | recvbuf           | address of receive buffer (choice) |
| IN   | recvcount         | number of elements in receive buffer (non-negative integer) |
| IN   | recvtype          | data type of receive buffer elements (handle) |
| IN   | root              | rank of sending process (integer) |
| IN   | comm              | communicator (handle) |

```c
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```c
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
        RECVTYPER, ROOT, COMM, IERROR)
```
5.6. SCATTER

The examples in this section use intracommunicators.

Example 5.11 The reverse of Example 5.2. Scatter sets of 100 ints from the root to each process in the group. See Figure 5.9.
CHAPTER 5. COLLECTIVE COMMUNICATION

Example 5.12 The reverse of Example 5.5. The root process scatters sets of 100 ints to the other processes, but the sets of 100 are stride ints apart in the sending buffer. Requires use of MPI_SCATTERV. Assume stride ≥ 100. See Figure 5.10.

```c
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Example 5.13 The reverse of Example 5.9. We have a varying stride between blocks at sending (root) side, at the receiving side we receive into the i-th column of a 100×150 C array. See Figure 5.11.

```c
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
displs = (int *)malloc(gsize*stride*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```
Figure 5.10: The root process scatters sets of 100 ints, moving by stride ints from send to send in the scatter.

int gsize, recvarray[100][150], *rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;

MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank);

stride = (int *)malloc(gsize*sizeof(int));
.../* stride[i] for i = 0 to gsize-1 is set somehow
   * sendbuf comes from elsewhere*/
...displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
displs[i] = offset;
offset += stride[i];
scounts[i] = 100 - i;
}

/* Create datatype for the column we are receiving*/
MPI_Type_vector( 100 - myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,
    root, comm);
CHAPTER 5. COLLECTIVE COMMUNICATION

Figure 5.11: The root scatters blocks of $100-i$ ints into column $i$ of a $100 \times 150$ C array. At the sending side, the blocks are $\text{stride}[i]$ ints apart.

5.7 Gather-to-all

MPI_ALLGATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

- IN sendbuf: starting address of send buffer (choice)
- IN sendcount: number of elements in send buffer (non-negative integer)
- IN sendtype: data type of send buffer elements (handle)
- OUT recvbuf: address of receive buffer (choice)
- IN recvcount: number of elements received from any process (non-negative integer)
- IN recvtype: data type of receive buffer elements (handle)
- IN comm: communicator (handle)

int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLGATHER can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the $j$-th process is received by every process and placed in the $j$-th block of the buffer recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process.
5.7. GATHER-TO-ALL

If comm is an intracommunicator, the outcome of a call to MPI_ALLGATHER(...) is as if all processes executed n calls to

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm),

for root = 0, ..., n-1. The rules for correct usage of MPI_ALLGATHER are easily found from the corresponding rules for MPI_GATHER.

The “in place” option for intracommunicators is specified by passing the value MPI_INPLACE to the argument sendbuf of all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

Advice to users. The communication pattern of MPI_ALLGATHER executed on an intercommunication domain need not be symmetric. The number of items sent by processes in group A (as specified by the arguments sendcount, sendtype in group A and the arguments recvcount, recvtype in group B), need not equal the number of items sent by processes in group B (as specified by the arguments sendcount, sendtype in group B and the arguments recvcount, recvtype in group A). In particular, one can move data in only one direction by specifying sendcount = 0 for the communication in the reverse direction.

(End of advice to users.)

MPI_ALLGATHERV( sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcounts non-negative integer array (of length group size) containing the number of elements that are received from each process
IN displs integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN recvtype data type of receive buffer elements (handle)
IN comm communicator (handle)

int MPI_allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, 
void* recvbuf, int *recvcounts, int *displs, 

MPI_ALLGATHERV( sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements in send buffer (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcounts non-negative integer array (of length group size) containing the number of elements that are received from each process
IN displs integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN recvtype data type of receive buffer elements (handle)
IN comm communicator (handle)

int MPI_allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, 
void* recvbuf, int *recvcounts, int *displs,
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)

MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes receive the result, instead of just the root. The block of data sent from the \( j \)-th process is received by every process and placed in the \( j \)-th block of the buffer recvbuf. These blocks need not all be the same size.

The type signature associated with sendcount, sendtype, at process \( j \) must be equal to the type signature associated with recvcounts\([j]\), recvtype at any other process.

If comm is an intracommunicator, the outcome is as if all processes executed calls to

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm),

for root = 0, ..., n-1. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV.

The “in place” option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV

The examples in this section use intracommunicators.

Example 5.14 The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

```c
MPI_Comm comm; int gsize, sendarray[100]; int *rbuf; ...
MPI_Comm_size( comm, &gsize); rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

After the call, every process has the group-wide concatenation of the sets of data.

Example 5.14 The all-gather version of Example 5.2. Using MPI_ALLGATHERV, we will gather 100 ints from every process in the group to every process.

```c
MPI_Datatype recvtype, MPI_Comm comm)
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)

MPI_ALLGATHERV can be thought of as MPI_GATHERV, but where all processes receive the result, instead of just the root. The block of data sent from the \( j \)-th process is received by every process and placed in the \( j \)-th block of the buffer recvbuf. These blocks need not all be the same size.

The type signature associated with sendcount, sendtype, at process \( j \) must be equal to the type signature associated with recvcounts\([j]\), recvtype at any other process.

If comm is an intracommunicator, the outcome is as if all processes executed calls to

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm),

for root = 0, ..., n-1. The rules for correct usage of MPI_ALLGATHERV are easily found from the corresponding rules for MPI_GATHERV.

The “in place” option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at all processes. sendcount and sendtype are ignored. Then the input data of each process is assumed to be in the area where that process would receive its own contribution to the receive buffer.

If comm is an intercommunicator, then each process in group A contributes a data item; these items are concatenated and the result is stored at each process in group B. Conversely the concatenation of the contributions of the processes in group B is stored at each process in group A. The send buffer arguments in group A must be consistent with the receive buffer arguments in group B, and vice versa.

5.7.1 Examples using MPI_ALLGATHER, MPI_ALLGATHERV

The examples in this section use intracommunicators.

Example 5.14 The all-gather version of Example 5.2. Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

```c
MPI_Comm comm; int gsize, sendarray[100]; int *rbuf; ...
MPI_Comm_size( comm, &gsize); rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

After the call, every process has the group-wide concatenation of the sets of data.
5.8 All-to-All Scatter/Gather

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN sendbuf starting address of send buffer (choice)
IN sendcount number of elements sent to each process (non-negative integer)
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcount number of elements received from any process (non-negative integer)
IN recvtype data type of receive buffer elements (handle)
IN comm communicator (handle)

int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, COMM, IERROR)

픙type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
void MPI::Comm::Alltoall(const void* sendbuf, int sendcount, const MPI::Datatype& sendtype, void* recvbuf, int recvcount, const MPI::Datatype& recvtype) const = 0

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the j-th block of recvbuf.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

If comm is an intracommunicator, the outcome is as if each process executed a send to each process (itself included) with a call to,

MPI_Send(sendbuf + i * sendcount * extent(sendtype), sendcount, sendtype, i, ...),

and a receive from every other process with a call to,

MPI_Recv(recvbuf + i * recvcount * extent(recvtype), recvcount, recvtype, i, ...).

All arguments on all processes are significant. The argument comm must have identical values on all processes.

No “in place” option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process
Advice to users. When all-to-all is executed on an intercommunication domain, then the number of data items sent from processes in group A to processes in group B need not equal the number of items sent in the reverse direction. In particular, one can have unidirectional communication by specifying $sendcount = 0$ in the reverse direction.

(End of advice to users.)

MPI\_\texttt{ALLTOALLV}(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN $sendbuf$ starting address of send buffer (choice)

IN $sendcounts$ non-negative integer array equal to the group size specifying the number of elements to send to each processor

IN $sdispls$ integer array (of length group size). Entry $j$ specifies the displacement (relative to $sendbuf$ from which to take the outgoing data destined for process $j$)

IN $sendtype$ data type of send buffer elements (handle)

OUT $recvbuf$ address of receive buffer (choice)

IN $recvcounts$ non-negative integer array equal to the group size specifying the number of elements that can be received from each processor

IN $rdispls$ integer array (of length group size). Entry $i$ specifies the displacement (relative to $recvbuf$ at which to place the incoming data from process $i$)

IN $recvtype$ data type of receive buffer elements (handle)

IN $comm$ communicator (handle)

int MPI\_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, MPI\_Datatype sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI\_Datatype recvtype, MPI\_Comm comm)

MPI\_\texttt{ALLTOALLV}(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS, RDISPLS, RECEVTYPE, COMM, MPI\_Comm)

\texttt{<type> SENDBUF(*)}, \texttt{RECVBUF(*)} INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, REVCOUNTS(*), RDISPLS(*), RECEVTYPE, COMM, IERROR

void MPI::Comm::Alltoallv(const void* sendbuf, const int sendcounts[], const int sdispls[], const MPI\_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], const MPI\_Datatype recvtype) const = 0

i in group $A$ should be consistent with the $i$-th receive buffer of process $j$ in group $B$, and vice versa.
5.8. All-to-All Scatter/Gather

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdisps and the location of the placement of the data on the receive side is specified by rdisps.

If comm is an intracommmunicator, then the j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. These blocks need not all have the same size.

The type signature associated with sendcount[i], sendtype at process i must be equal to the type signature associated with recvcount[j], recvtype at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. Distinct type maps between sender and receiver are still allowed.

The outcome is as if each process sent a message to every other process with,

MPI_Send(sendbuf + disp[i] · extent(sendtype), sendcounts[i], sendtype, i, ...),

and received a message from every other process with a call to

MPI_Recv(recvbuf + disp[i] · extent(recvtype), recvcounts[i], recvtype, i, ...).

All arguments on all processes are significant. The argument comm must have identical values on all processes.

No “in place” option is supported.

If comm is an intercommunicator, then the outcome is as if each process in group A sends a message to each process in group B, and vice versa. The j-th send buffer of process i in group A should be consistent with the i-th receive buffer of process j in group B, and vice versa.

Rationale. The definitions of MPI_ALLTOALL and MPI_ALLTOALLV give as much flexibility as one would achieve by specifying an independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (End of rationale.)

Advice to implementors. Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (End of advice to implementors.)
MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recv
types, comm)

IN sendbuf
starting address of send buffer (choice)

IN sendcounts
integer array equal to the group size specifying the
number of elements to send to each processor (array
of non-negative integers)

IN sdispls
integer array (of length group size). Entry j specifies
the displacement in bytes (relative to sendbuf) from
which to take the outgoing data destined for process
j (array of integers)

IN sendtypes
array of datatypes (of length group size). Entry j specifies
the type of data to send to process j (array
of handles)

OUT recvbuf
address of receive buffer (choice)

IN recvcounts
integer array equal to the group size specifying thenumber of elements that can be received from each
processor (array of non-negative integers)

IN rdispls
integer array (of length group size). Entry i specifies
the displacement in bytes (relative to recvbuf) at which
to place the incoming data from process i (array
of integers)

IN recvtypes
array of datatypes (of length group size). Entry i specifies
the type of data received from process i (array
of handles)

IN comm
communicator (handle)

int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)

MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
RDISPLS, RECVTYPES, COMM, IERROR)

$type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
RDISPLS(*), RECVTYPES(*), COMM, IERROR

void MPI::Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
const int sdispls[], const MPI::Datatype sendtypes[], void* recvbuf,
const int recvcounts[], const int rdispls[], const
MPI::Datatype recvtypes[]) const = 0

MPI_ALLTOALLW is the most general form of All-to-all. Like
MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-
allows separate specification of count, displacement and datatype. In addition, to allow max-
imum flexibility, the displacement of blocks within the send and receive buffers is specified
in bytes.

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recv
types, comm)

IN sendbuf
starting address of send buffer (choice)

IN sendcounts
integer array equal to the group size specifying the
number of elements to send to each processor (array
of non-negative integers)

IN sdispls
integer array (of length group size). Entry j specifies
the displacement in bytes (relative to sendbuf) from
which to take the outgoing data destined for process
j (array of integers)

IN sendtypes
array of datatypes (of length group size). Entry j specifies
the type of data to send to process j (array
of handles)

OUT recvbuf
address of receive buffer (choice)

IN recvcounts
integer array equal to the group size specifying thenumber of elements that can be received from each
processor (array of non-negative integers)

IN rdispls
integer array (of length group size). Entry i specifies
the displacement in bytes (relative to recvbuf) at which
to place the incoming data from process i (array
of integers)

IN recvtypes
array of datatypes (of length group size). Entry i specifies
the type of data received from process i (array
of handles)

IN comm
communicator (handle)

int MPI_Alltoallw(void *sendbuf, int sendcounts[], int sdispls[],
MPI_Datatype sendtypes[], void *recvbuf, int recvcounts[],
int rdispls[], MPI_Datatype recvtypes[], MPI_Comm comm)

MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
RDISPLS, RECVTYPES, COMM, IERROR)

$type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
RDISPLS(*), RECVTYPES(*), COMM, IERROR

void MPI::Comm::Alltoallw(const void* sendbuf, const int sendcounts[],
const int sdispls[], const MPI::Datatype sendtypes[], void* recvbuf,
const int recvcounts[], const int rdispls[], const
MPI::Datatype recvtypes[]) const = 0

MPI_ALLTOALLW is the most general form of All-to-all. Like
MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW al-
allows separate specification of count, displacement and datatype. In addition, to allow max-
imum flexibility, the displacement of blocks within the send and receive buffers is specified
in bytes.
5.9 Global Reduction Operations

The functions in this section perform a global reduce operation (such as sum, max, logical AND, etc.) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.
5.9.1 Reduce

MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)

IN sendbuf address of send buffer (choice)
OUT recvbuf address of receive buffer (choice, significant only at root)
IN count number of elements in send buffer (non-negative integer)
IN datatype data type of elements of send buffer (handle)
IN op reduce operation (handle)
IN root rank of root process (integer)
IN comm communicator (handle)

int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
                        const MPI::Datatype& datatype, const MPI::Op& op, int root)
    const = 0

If comm is an intracommunicator, MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type.

The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then recvbuf(1) = global max(sendbuf(1)) and recvbuf(2) = global max(sendbuf(2)).

Section 5.9.2, lists the set of predefined operations provided by MPI. That section also enumerates the datatypes each operation can be applied to. In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 5.9.5.

The operation op is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative. The “canonical” evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation.
5.9. GLOBAL REDUCTION OPERATIONS

This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors. (End of advice to implementors.)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 5.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. Safest is to ensure that the same function is passed to MPI_REDUCE by each process. (End of advice to users.)

Overlapping datatypes are permitted in “send” buffers. Overlapping datatypes in “receive” buffers are erroneous and may give unpredictable results.

The “in place” option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

5.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. These operations are invoked by placing the following in op.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bit-wise and</td>
</tr>
</tbody>
</table>

This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

Advice to implementors. It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of processors. (End of advice to implementors.)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 5.9.2 and Section 5.9.4. Furthermore, the datatype and op given for predefined operators must be the same on all processes.

Note that it is possible for users to supply different user-defined operations to MPI_REDUCE in each process. MPI does not define which operations are used on which operands in this case. User-defined operators may operate on general, derived datatypes. In this case, each argument that the reduce operation is applied to is one element described by such a datatype, which may contain several basic values. This is further explained in Section 5.9.5.

Advice to users. Users should make no assumptions about how MPI_REDUCE is implemented. Safest is to ensure that the same function is passed to MPI_REDUCE by each process. (End of advice to users.)

Overlapping datatypes are permitted in “send” buffers. Overlapping datatypes in “receive” buffers are erroneous and may give unpredictable results.

The “in place” option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the call involves all processes in the intercommunicator, but with one group (group A) defining the root process. All processes in the other group (group B) pass the same value in argument root, which is the rank of the root in group A. The root passes the value MPI_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Only send buffer arguments are significant in group B and only receive buffer arguments are significant at the root.

5.9.2 Predefined Reduction Operations

The following predefined operations are supplied for MPI_REDUCE and related functions MPI_ALLREDUCE, MPI_REDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. These operations are invoked by placing the following in op.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bit-wise and</td>
</tr>
</tbody>
</table>
MPI_LOR
MPI_LXOR
MPI_BXOR
MPI_MAXLOC
MPI_MINLOC

The two operations MPI_MINLOC and MPI_MAXLOC are discussed separately in Section 5.9.4. For the other predefined operations, we enumerate below the allowed combinations of op and datatype arguments. First, define groups of MPI basic datatypes in the following way:

C integer: MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_LONG_LONG_INT, MPI_LONG_LONG (as synonym), MPI_UNSIGNED_LONG, MPI_SIGNED_CHAR, MPI_UNSIGNED_CHAR
Fortran integer: MPI_INTEGER
Floating point: MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION
Logical: MPI_LOGICAL
Complex: MPI_COMPLEX
Byte: MPI_BYTE

Now, the valid datatypes for each option is specified below.

MPL_MAX, MPL_MIN
MPL_SUM, MPL_PROD
MPL_BAND, MPL_BOR, MPL_BXOR

The following examples use intracommunicators.

Example 5.15 A routine that computes the dot product of two vectors that are distributed across a group of processes and returns the answer at node zero.

A routine that computes the dot product of two vectors that are distributed across a group of processes and returns the answer at node zero.

SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
REAL a(m), b(m) ! local slice of array
REAL c ! result (at node zero)
REAL sum
INTEGER m, comm, i, ierr
! local sum
sum = 0.0

The following examples use intracommunicators.

Example 5.15 A routine that computes the dot product of two vectors that are distributed across a group of processes and returns the answer at node zero.

SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
REAL a(m), b(m) ! local slice of array
REAL c ! result (at node zero)
REAL sum
INTEGER m, comm, i, ierr
! local sum
sum = 0.0
5.9. GLOBAL REDUCTION OPERATIONS

DO i = 1, m
    sum = sum + a(i) * b(i)
END DO

! global sum
CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
RETURN

Example 5.16 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at node zero.

SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
    REAL a(m), b(m,n) ! local slice of array
    REAL c(n) ! result
    REAL sum(n)!
    INTEGER n, comm, i, j, ierr

    ! local sum
    DO j= 1, n
        sum(j) = 0.0
        DO i = 1, m
            sum(j) = sum(j) + a(i) * b(i,j)
        END DO
    END DO

    ! global sum
    CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)

    ! return result at node zero (and garbage at the other nodes)
RETURN

5.9.3 Signed Characters and Reductions

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR (which represents printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR and MPI_WCHAR will be translated so as to preserve the printable character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

Advice to users. The types MPI_CHAR and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (End of advice to users.)

Advice to users. The types MPI_CHAR and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (End of advice to users.)
5.9.4 MINLOC and MAXLOC

The operator MPI_MINLOC is used to compute a global minimum and also an index attached to the minimum value. MPI_MAXLOC similarly computes a global maximum and index. One application of these is to compute a global minimum (maximum) and the rank of the process containing this value.

The operation that defines MPI_MAXLOC is:
\[
\left( \begin{array}{c} u \\ i \end{array} \right) \circ \left( \begin{array}{c} v \\ j \end{array} \right) = \left( \begin{array}{c} w \\ k \end{array} \right)
\]

where
\[
w = \max(u, v)
\]
and
\[
k = \begin{cases} 
  i & \text{if } u > v \\
  \min(i, j) & \text{if } u = v \\
  j & \text{if } u < v
\end{cases}
\]

MPI_MINLOC is defined similarly:
\[
\left( \begin{array}{c} u \\ i \end{array} \right) \circ \left( \begin{array}{c} v \\ j \end{array} \right) = \left( \begin{array}{c} w \\ k \end{array} \right)
\]

where
\[
w = \min(u, v)
\]
and
\[
k = \begin{cases} 
  i & \text{if } u < v \\
  \min(i, j) & \text{if } u = v \\
  j & \text{if } u > v
\end{cases}
\]

Both operations are associative and commutative. Note that if MPI_MAXLOC is applied to reduce a sequence of pairs \((u_0, 0), (u_1, 1), \ldots, (u_{n-1}, n-1)\), then the value returned is \((u, r)\), where \(u = \max u_i\) and \(r\) is the index of the first global maximum in the sequence. Thus, if each process supplies a value and its rank within the group, then a reduce operation is defined to operate on arguments that consist of a pair: value and index. For both Fortran and C, types are provided to describe the pair. The potentially mixed-type nature of such arguments is a problem in Fortran. The problem is circumvented, for Fortran, by having the MPI-provided type consist of a pair of the same type as value, and coercing the index to this type also. In C, the MPI-provided pair type has distinct types and the index is an int.

In order to use MPI_MINLOC and MPI_MAXLOC in a reduce operation, one must provide a datatype argument that represents a pair (value and index). MPI provides nine such

The operation that defines MPI_MAXLOC is:
\[
\left( \begin{array}{c} u \\ i \end{array} \right) \circ \left( \begin{array}{c} v \\ j \end{array} \right) = \left( \begin{array}{c} w \\ k \end{array} \right)
\]

where
\[
w = \max(u, v)
\]
and
\[
k = \begin{cases} 
  i & \text{if } u > v \\
  \min(i, j) & \text{if } u = v \\
  j & \text{if } u < v
\end{cases}
\]

MPI_MINLOC is defined similarly:
\[
\left( \begin{array}{c} u \\ i \end{array} \right) \circ \left( \begin{array}{c} v \\ j \end{array} \right) = \left( \begin{array}{c} w \\ k \end{array} \right)
\]

where
\[
w = \min(u, v)
\]
and
\[
k = \begin{cases} 
  i & \text{if } u < v \\
  \min(i, j) & \text{if } u = v \\
  j & \text{if } u > v
\end{cases}
\]
predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with each of the following datatypes.

### Fortran:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_2REAL</td>
<td>pair of REALs</td>
</tr>
<tr>
<td>MPI_2DOUBLE_PRECISION</td>
<td>pair of DOUBLE PRECISION variables</td>
</tr>
<tr>
<td>MPI_2INTEGER</td>
<td>pair of INTEGERS</td>
</tr>
</tbody>
</table>

### C:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_FLOAT_INT</td>
<td>float and int</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>double and int</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>long and int</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>short and int</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>long double and int</td>
</tr>
</tbody>
</table>

The datatype MPI_2REAL is as if defined by the following (see Section 4.1).

`MPI_TYPE_CONTIGUOUS(2, MPI_REAL, MPI_2REAL)`

Similar statements apply for MPI_2INTEGER, MPI_2DOUBLE_PRECISION, and MPI_2INT.

The datatype MPI_FLOAT_INT is as if defined by the following sequence of instructions.

```c
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)block[0] = 1block[1] = 1MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

Similar statements apply for MPI_LONG_INT and MPI_DOUBLE_INT.

The following examples use intracommunicators.

#### Example 5.17

Each process has an array of 30 `double` s, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```c
// each process has an array of 30 double: ain[30]
 double ain[30], aout[30];
 int ind[30];
 struct {
   double val;
   int rank;
 } in[30], out[30];
 int i, myrank, root;
```
MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i) {
in[i].val = ain[i];
in[i].rank = myrank;
}
MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);

/* At this point, the answer resides on process root */
if (myrank == root) {
    /* read ranks out */
    for (i=0; i<30; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
    }
}

Example 5.18 Same example, in Fortran.

! each process has an array of 30 double: ain(30)
DOUBLE PRECISION ain(30), aout(30)
INTEGER ind(30)
DOUBLE PRECISION in(2,30), out(2,30)
INTEGER i, myrank, root, ierr
CALL MPI_COMM_RANK(comm, myrank, ierr)
DO I=1, 30
    in(1,i) = ain(i)
in(2,i) = myrank ! myrank is coerced to a double
END DO
CALL MPI_REDUCE(in, out, 30, MPI_DOUBLE_PRECISION, MPI_MAXLOC, root, comm, ierr)

! At this point, the answer resides on process root
IF (myrank .EQ. root) THEN
    ! read ranks out
    DO I=1, 30
        aout(i) = out(1,i)
        ind(i) = out(2,i) ! rank is coerced back to an integer
    END DO
END IF

Example 5.19 Each process has a non-empty array of values. Find the minimum global value, the rank of the process that holds it and its index on this process.
#define LEN 1000

float val[LEN]; /* local array of values */
int count; /* local number of values */
int myrank, minrank, minindex;
float minval;

struct {
   float value;
   int index;
} in, out;

/* local minloc */
in.value = val[0];
in.index = 0;
for (i=1; i < count; i++)
   if (in.value > val[i]) {
      in.value = val[i];
in.index = i;
   }

/* global minloc */
MPI_Comm_rank(comm, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce( in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );

/* At this point, the answer resides on process root */
if (myrank == root) {
   /* read answer out */
   minval = out.value;
   minrank = out.index / LEN;
   minindex = out.index % LEN;
}

Rationale. The definition of MPI_MINLOC and MPI_MAXLOC given here has the advantage that it does not require any special-case handling of these two operations: they are handled like any other reduce operation. A programmer can provide his or her own definition of MPI_MAXLOC and MPI_MINLOC, if so desired. The disadvantage is that values and indices have to be first interleaved, and that indices and values have to be coerced to the same type, in Fortran. (End of rationale.)
5.9.5 User-Defined Reduction Operations

```c
MPI_Op_create(function, commute, op)
```

The `MPI_Op_create` function creates a user-defined global operation `op` based on the provided user-defined function `function` and whether the operation is commutative (`commute`). This `op` can subsequently be used in `MPI_REDUCE`, `MPI_ALLREDUCE`, `MPI_REDUCE_SCATTER`, `MPI_SCAN`, and `MPI_EXSCAN`. The user-defined operation is assumed to be associative. If `commute = true`, the order of evaluation can be changed, taking advantage of the commutativity and associativity of the operation. If `commute = false`, then the order of evaluation can be changed, taking advantage of the associativity of the operation.

The user-defined function must have the following arguments:

- `invec`, `inoutvec`, `len`, and `datatype`.

The ISO C prototype for the function is the following:

```c
typedef void MPI_User_function(void *invec, void *inoutvec, int *len, MPI_Datatype *datatype);
```

The Fortran declaration of the user-defined function appears below:

```fortran
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
  TYPE(INVEC(len)), INOUTVEC(len)
  INTEGER LEN, TYPE
END SUBROUTINE USER_FUNCTION
```

The C++ declaration of the user-defined function appears below:

```c
typedef void MPI::User_function(const void* invec, void *inoutvec, int len, const MPI_Datatype* datatype);
```

The `datatype` argument is a handle to the data type that was passed into the call to `MPI_REDUCE`. The user reduce function should be written such that the following holds:

- Let `u[0], ..., u[len-1]` be the `len` elements in the communication buffer described by the arguments `invec`, `len`, and `datatype` when the function is invoked; let `v[0], ..., v[len-1]` be `len` elements in the communication buffer described by the arguments `inoutvec`, `len`, and `datatype` when the function is invoked; let `w[0], ..., w[len-1]` be `len` elements in the communication buffer described by the arguments `inoutvec`, `len`, and `datatype` when the function returns; then `w[i] = u[i] \odot v[i]`, for `i = 0, ..., len-1`, where `\odot` is the reduce operation that the function computes.
Informally, we can think of \texttt{invec} and \texttt{inoutvec} as arrays of \textit{len} elements that function \texttt{combine} results in. The result of the reduction overwrites values in \texttt{inoutvec}, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on \textit{len} elements: i.e., the function returns \texttt{inoutvec[i]} = \texttt{invec[i] \circ inoutvec[i]}, for \( i = 0, \ldots, \text{count} - 1 \), where \( \circ \) is the combining operation computed by the function.

**Rationale.** The \texttt{len} argument allows \texttt{MPI\_REDUCE} to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the \texttt{datatype} argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. \texttt{MPI\_ABORT} may be called inside the function in case of an error.

**Advice to users.** Suppose one defines a library of user-defined reduce functions that are overloaded: the \texttt{datatype} argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot “decode” the \texttt{datatype} argument that it is passed, and cannot identify, by itself, the correspondence between the \texttt{datatype} handles and the \texttt{datatype} they represent. This correspondence was established when the \texttt{datatypes} were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the \texttt{datatypes} that are used by the library, and store handles to these \texttt{datatypes} in global, static variables that are shared by the user code and the library code.

The Fortran version of \texttt{MPI\_REDUCE} will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type \texttt{datatype} argument; the C version will use C calling convention and the C representation of a \texttt{datatype} handle.

Users who plan to mix languages should define their reduction functions accordingly. (*End of advice to users.*)

**Advice to implementors.** We outline below a naive and inefficient implementation of \texttt{MPI\_REDUCE} not supporting the “in place” option.

\begin{verbatim}
MPI_Comm_size(comm, &groupsize);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1, ...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root */
\end{verbatim}

Informally, we can think of \texttt{invec} and \texttt{inoutvec} as arrays of \textit{len} elements that function \texttt{combine} results in. The result of the reduction overwrites values in \texttt{inoutvec}, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on \textit{len} elements: i.e., the function returns \texttt{inoutvec[i]} = \texttt{invec[i] \circ inoutvec[i]}, for \( i = 0, \ldots, \text{count} - 1 \), where \( \circ \) is the combining operation computed by the function.

**Rationale.** The \texttt{len} argument allows \texttt{MPI\_REDUCE} to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the \texttt{datatype} argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different data types. (*End of rationale.*)

General datatypes may be passed to the user function. However, use of datatypes that are not contiguous is likely to lead to inefficiencies.

No MPI communication function may be called inside the user function. \texttt{MPI\_ABORT} may be called inside the function in case of an error.

**Advice to users.** Suppose one defines a library of user-defined reduce functions that are overloaded: the \texttt{datatype} argument is used to select the right execution path at each invocation, according to the types of the operands. The user-defined reduce function cannot “decode” the \texttt{datatype} argument that it is passed, and cannot identify, by itself, the correspondence between the \texttt{datatype} handles and the \texttt{datatype} they represent. This correspondence was established when the \texttt{datatypes} were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the \texttt{datatypes} that are used by the library, and store handles to these \texttt{datatypes} in global, static variables that are shared by the user code and the library code.

The Fortran version of \texttt{MPI\_REDUCE} will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type \texttt{datatype} argument; the C version will use C calling convention and the C representation of a \texttt{datatype} handle.

Users who plan to mix languages should define their reduction functions accordingly. (*End of advice to users.*)

**Advice to implementors.** We outline below a naive and inefficient implementation of \texttt{MPI\_REDUCE} not supporting the “in place” option.

\begin{verbatim}
MPI_Comm_size(comm, &groupsize);
MPI_Comm_rank(comm, &rank);
if (rank > 0) {
    MPI_Recv(tempbuf, count, datatype, rank-1, ...);
    User_reduce(tempbuf, sendbuf, count, datatype);
}
if (rank < groupsize-1) {
    MPI_Send(sendbuf, count, datatype, rank+1, ...);
}
/* answer now resides in process groupsize-1 ... now send to root */
\end{verbatim}
The reduction computation proceeds, sequentially, from process 0 to process groupsize-1. This order is chosen so as to respect the order of a possibly non-commutative operator defined by the function User_reduction() of the function. A more efficient implementation is achieved by taking advantage of associativity and using a logarithmic tree reduction. Commutativity can be used to advantage, for those cases in which the commute argument to MPI_OP_CREATE is true. Also, the amount of temporary buffer required can be reduced, and communication can be pipelined with computation, by transferring and reducing the elements in chunks of size \( \text{len} < \text{count} \).

The predefined reduce operations can be implemented as a library of user-defined operations. However, better performance might be achieved if MPI_REDUCE handles these functions as a special case. (End of advice to implementors.)

Example of User-defined Reduce
It is time for an example of user-defined reduction. The example in this section uses an intracommunicator.

Example 5.20 Compute the product of an array of complex numbers, in C.

typedef struct {
    double real, imag;
} Complex;

/* the user-defined function
void myProd( Complex *in, Complex *inout, int *len, MPI_Datatype *dptr )
{
    int i;
    Complex c;
    for (i=0; i< *len; ++i) {
        c.real = inout->real*in->real -
                inout->imag*in->imag;
        c.imag = inout->real*in->imag +
                inout->imag*in->real;
        *inout = c;
        inout++;in++;
    }
}

...
MPI\_ALLREDUCE( sendbuf, recvbuf, count, datatype, op, comm)

IN  sendbuf  starting address of send buffer (choice)

OUT recvbuf  starting address of receive buffer (choice)

IN  count  number of elements in send buffer (non-negative integer)

IN  datatype  data type of elements of send buffer (handle)

IN  op  operation (handle)

IN  comm  communicator (handle)

int MPI\_Allreduce(void* sendbuf, void* recvbuf, int count,
                   MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm)

MPI\_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)

INTEGER COUNT, DATATYPE, OP, COMM, IERROR

void MPI::Comm::Allreduce(const void* sendbuf, void* recvbuf, int count,
                          const MPI\::Datatype& datatype, const MPI\::Op& op) const = 0

If comm is an intracommunicator, MPI\_ALLREDUCE behaves the same as
MPI\_REDUCE except that the result appears in the receive buffer of all the group members.

Advice to implementors. The all-reduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance. (End of advice to implementors.)

The “in place” option for intracommunicators is specified by passing the value MPI\_IN\_PLACE to the argument sendbuf at all processes. In this case, the input data is taken at each process from the receive buffer, where it will be replaced by the output data.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in group A is stored at each process in group B, and vice versa. Both groups should provide count and datatype arguments that specify the same type signature.

The following example uses an intracommunicator.

Example 5.21 A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).

SUBROUTINE PAR\_BLAS2(m, n, a, b, c, comm)
REAL a(m), b(m,n) ! local slice of array
REAL c(m) ! result
REAL sum(n)
INTEGER n, comm, i, j, ierr
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
5.10. REDUCE-SCATTER

sum(j) = sum(j) + a(i) * b(i,j)
END DO
END DO

! global sum
CALL MPI_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

! return result at all nodes
RETURN

5.10 Reduce-Scatter

MPI includes a variant of the reduce operations where the result is scattered to all processes in a group on return.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

| IN  | sendbuf                          | starting address of send buffer (choice) |
| OUT | recvbuf                          | starting address of receive buffer (choice) |
| IN  | recvcounts                       | non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes. |
| IN  | datatype                         | data type of elements of input buffer (handle) |
| IN  | op                               | operation (handle) |
| IN  | comm                             | communicator (handle) |

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

void MPI::Comm::Reduce_scatter(const void* sendbuf, void* recvbuf, int recvcounts[], const MPI::Datatype& datatype, const MPI::Op& op) const = 0

If comm is an intracommunicator, MPI_REDUCE_SCATTER first does an element-wise reduction on vector of count = \sum recvcounts[i] elements in the send buffer defined by sendbuf, count and datatype. Next, the resulting vector of results is split into n disjoint segments, where n is the number of members in the group. Segment i contains recvcounts[i] elements. The i-th segment is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

Advice to implementors. The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] elements on each process.
The “in place” option for intracommitters is specified by passing `MPI_IN_PLACE` in the `sendbuf` argument. In this case, the input data is taken from the top of the receive buffer.

If `comm` is an intercommunicator, then the result of the reduction of the data provided by processes in group A is scattered among processes in group B, and vice versa. Within each group, all processes provide the same `recvcounts` argument, and the sum of the `recvcounts` entries should be the same for the two groups.

**Rationale.** The last restriction is needed so that the length of the send buffer can be determined by the sum of the local `recvcounts` entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

### 5.11 Scan

#### 5.11.1 Inclusive Scan

```c
MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm )
```

<table>
<thead>
<tr>
<th>IN</th>
<th>sendbuf</th>
<th>starting address of send buffer (choice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>recvbuf</td>
<td>starting address of receive buffer (choice)</td>
</tr>
<tr>
<td>IN</td>
<td>count</td>
<td>number of elements in input buffer (non-negative integer)</td>
</tr>
<tr>
<td>IN</td>
<td>datatype</td>
<td>data type of elements of input buffer (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>op</td>
<td>operation (handle)</td>
</tr>
<tr>
<td>IN</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
</tbody>
</table>

int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )

**MPI_SCAN** is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank `i`, the reduction of the values in the send buffers of processes with ranks `0,...,i` (inclusive). The type of operations supported, their semantics, and the constraints on send and receive buffers are as for `MPI_REDUCE`.

The “in place” option for intracommitters is specified by passing `MPI_IN_PLACE` in the `sendbuf` argument. In this case, the input data is taken from the receive buffer, and replaced by the output data.
5.11. SCAN

This operation is invalid for intercommunicators.

5.11.2 Exclusive Scan

MPI_EXSCAN(sendbuf, recvbuf, count, datatype, op, comm)

| IN  | sendbuf       | starting address of send buffer (choice) |
| OUT | recvbuf      | starting address of receive buffer (choice) |
| IN  | count        | number of elements in input buffer (non-negative integer) |
| IN  | datatype     | data type of elements of input buffer (handle) |
| IN  | op           | operation (handle) |
| IN  | comm         | intracommunicator (handle) |

int MPI_Exscan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)

In a communicator, MPI_EXSCAN is used to perform a prefix reduction on data distributed across the group. The value in recvbuf on the process with rank 0 is undefined, and recvbuf is not significant on process 0. The value in recvbuf on the process with rank 1 is defined as the value in sendbuf on the process with rank 0. For processes with rank \( i > 1 \), the operation returns, in the receive buffer of the process with rank \( i \), the reduction of the values in the send buffers of processes with ranks 0, ..., \( i - 1 \) (inclusive).

The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

No “in place” option is supported.

This operation is invalid for intercommunicators.

Advice to users. As for MPI_SCAN, MPI does not specify which processes may call the operation, only that the result be correctly computed. In particular, note that the process with rank 1 need not call the MPI_Op, since all it needs to do is to receive the value from the process with rank 0. However, all processes, even the processes with ranks zero and one, must provide the same op. (End of advice to users.)

Rationale. The exclusive scan is more general than the inclusive scan. Any inclusive scan operation can be achieved by using the exclusive scan and then locally combining the local contribution. Note that for non-invertable operations such as MPI_MAX, the exclusive scan cannot be computed with the inclusive scan.

No in-place version is specified for MPI_EXSCAN because it is not clear what this means for the process with rank zero. (End of rationale.)
5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

Example 5.22 This example uses a user-defined operation to produce a segmented scan.

A segmented scan takes, as input, a set of values and a set of logicals, and the logicals delineate the various segments of the scan. For example:

values: \(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\)

logicals: \(0, 0, 1, 1, 1, 0, 0, 1\)

result: \(v_1, v_1 + v_2, v_3 + v_4 + v_5, v_6 + v_7, v_8\)

The operator that produces this effect is,

\[
\left( \begin{array}{c}
  u \\
  i \\
\end{array} \right) \circ \left( \begin{array}{c}
  v \\
  j \\
\end{array} \right) = \left( \begin{array}{c}
  w \\
  j \\
\end{array} \right),
\]

where,

\[
w = \begin{cases} 
  u + v & \text{if } i = j \\
  v & \text{if } i \neq j
\end{cases}.
\]

Note that this is a non-commutative operator. C code that implements it is given below.

```c
typedef struct {
  double val;
  int log;
} SegScanPair;

/* the user-defined function */
void segScan( SegScanPair *in, SegScanPair *inout, int *len,
             MPI_Datatype *dptr ) {
  int i;
  SegScanPair c;
  for (i=0; i<*len; ++i) {
    if ( in->log == inout->log )
      c.val = in->val + inout->val;
    else
      c.val = inout->val;
    c.log = inout->log;
    *inout = c;
    inout++;}
}
```

Note that the inout argument to the user-defined function corresponds to the right-hand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.
5.12 Correctness

A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not. The following examples illustrate dangerous use of collective routines on intracommunicators.

**Example 5.23** The following is erroneous.

```c
switch(rank) {
  case 0:
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Bcast(buf2, count, type, 1, comm);
    break;
  case 1:
    MPI_Bcast(buf2, count, type, 1, comm);
    MPI_Bcast(buf1, count, type, 0, comm);
    break;
}
```

We assume that the group of `comm` is `{0, 1}`. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

**Example 5.24** The following is erroneous.

```c
switch(rank) {
  case 0:
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Bcast(buf2, count, type, 1, comm);
    break;
  case 1:
    MPI_Bcast(buf2, count, type, 1, comm);
    MPI_Bcast(buf1, count, type, 0, comm);
    break;
}
```

We assume that the group of `comm` is `{0, 1}`. Two processes execute two broadcast operations in reverse order. If the operation is synchronizing then a deadlock will occur.

Collective operations must be executed in the same order at all members of the communication group.

**Example 5.24** The following is erroneous.
switch(rank) {
  case 0:
    MPI_Bcast(buf1, count, type, 0, comm0);
    MPI_Bcast(buf2, count, type, 2, comm2);
    break;
  case 1:
    MPI_Bcast(buf1, count, type, 1, comm1);
    MPI_Bcast(buf2, count, type, 0, comm0);
    break;
  case 2:
    MPI_Bcast(buf1, count, type, 2, comm2);
    MPI_Bcast(buf2, count, type, 1, comm1);
    break;
}

Assume that the group of comm0 is {0, 1}, of comm1 is {1, 2} and of comm2 is {2, 0}. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependences occur.

Example 5.25 The following is erroneous.

switch(rank) {
  case 0:
    MPI_Bcast(buf1, count, type, 0, comm); MPI_Send(buf2, count, type, 1, tag, comm); break;
  case 1:
    MPI_Recv(buf2, count, type, 0, tag, comm, status); MPI_Bcast(buf1, count, type, 0, comm); break;
  case 2:
    MPI_Bcast(buf1, count, type, 2, comm2); MPI_Bcast(buf2, count, type, 1, comm1); break;
}

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero may block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Example 5.26 An unsafe, non-deterministic program.

switch(rank) {
  case 0:
    MPI_Bcast(buf1, count, type, 0, comm0);
    MPI_Bcast(buf2, count, type, 2, comm2);
    break;
  case 1:
    MPI_Bcast(buf1, count, type, 1, comm1);
    MPI_Bcast(buf2, count, type, 0, comm0);
    break;
  case 2:
    MPI_Bcast(buf1, count, type, 2, comm2);
    MPI_Bcast(buf2, count, type, 1, comm1);
    break;
}

Assume that the group of comm0 is {0, 1}, of comm1 is {1, 2} and of comm2 is {2, 0}. If the broadcast is a synchronizing operation, then there is a cyclic dependency: the broadcast in comm2 completes only after the broadcast in comm0; the broadcast in comm0 completes only after the broadcast in comm1; and the broadcast in comm1 completes only after the broadcast in comm2. Thus, the code will deadlock.

Collective operations must be executed in an order so that no cyclic dependences occur.

Example 5.26 An unsafe, non-deterministic program.
5.12. CORRECTNESS

First Execution

process: 0 1 2

recv broadcast broadcast broadcast send

Second Execution

broadcast send match recv

broadcastrecv send

Figure 5.12: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

MPI_Bcast(buf1, count, type, 0, comm);
MPI_Send(buf2, count, type, 1, tag, comm); break;

case 1:
    MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
    MPI_Bcast(buf1, count, type, 0, comm);
    break;

case 2:
    MPI_Send(buf2, count, type, 1, tag, comm);
    MPI_Bcast(buf1, count, type, 0, comm);
    break;
}

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 5.12. Note that the second execution has the peculiar effect that a send executed after the broadcast is received at another node before the broadcast. This example illustrates the fact that one should not rely on collective communication functions to have particular synchronization effects. A program that works correctly only when the first execution occurs (only when broadcast is synchronizing) is erroneous.

Finally, in multithreaded implementations, one can have more than one, concurrently executing, collective communication call at a process. In these situations, it is the user’s re-
sponsibility to ensure that the same communicator is not used concurrently by two different collective communication calls at the same process.

Advice to implementors. Assume that broadcast is implemented using point-to-point MPI communication. Suppose the following two rules are followed.

1. All receives specify their source explicitly (no wildcards).
2. Each process sends all messages that pertain to one collective call before sending any message that pertain to a subsequent collective call.

Then, messages belonging to successive broadcasts cannot be confused, as the order of point-to-point messages is preserved. It is the implementor’s responsibility to ensure that point-to-point messages are not confused with collective messages. One way to accomplish this is, whenever a communicator is created, to also create a “hidden communicator” for collective communication. One could achieve a similar effect more cheaply, for example, by using a hidden tag or context bit to indicate whether the communicator is used for point-to-point or collective communication. (End of advice to implementors.)
Chapter 6

Groups, Contexts, Communicators, and Caching

6.1 Introduction

This chapter introduces MPI features that support the development of parallel libraries. Parallel libraries are needed to encapsulate the distracting complications inherent in parallel implementations of key algorithms. They help to ensure consistent correctness of such procedures, and provide a "higher level" of portability than MPI itself can provide. As such, libraries prevent each programmer from repeating the work of defining consistent data structures, data layouts, and methods that implement key algorithms (such as matrix operations). Since the best libraries come with several variations on parallel systems (different data layouts, different strategies depending on the size of the system or problem, or type of floating point), this too needs to be hidden from the user.

We refer the reader to [42] and [3] for further information on writing libraries in MPI, using the features described in this chapter.

6.1.1 Features Needed to Support Libraries

The key features needed to support the creation of robust parallel libraries are as follows:

- Safe communication space, that guarantees that libraries can communicate as they need to, without conflicting with communication extraneous to the library,
- Group scope for collective operations, that allow libraries to avoid unnecessarily synchronizing uninvolved processes (potentially running unrelated code),
- Abstract process naming to allow libraries to describe their communication in terms suitable to their own data structures and algorithms,
- The ability to "adorn" a set of communicating processes with additional user-defined attributes, such as extra collective operations. This mechanism should provide a means for the user or library writer effectively to extend a message-passing notation.

In addition, a unified mechanism or object is needed for conveniently denoting communication context, the group of communicating processes, to house abstract process naming, and to store adornments.
6.1.2 MPI’s Support for Libraries

The corresponding concepts that MPI provides, specifically to support robust libraries, are as follows:

- **Contexts** of communication,
- **Groups** of processes,
- **Virtual topologies**,
- **Attribute caching**.

**Communicators.** (see [19, 40, 45]) encapsulate all of these ideas in order to provide the appropriate scope for all communication operations in MPI. Communicators are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes.

**Caching.** Communicators (see below) provide a “caching” mechanism that allows one to associate new attributes with communicators, on a par with MPI built-in features. This can be used by advanced users to adorn communicators further, and by MPI to implement some communicator functions. For example, the virtual-topology functions described in Chapter 7 are likely to be supported this way.

**Groups.** Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in MPI, but only communicators can be used in communication operations.

**Intra-communicators.** The most commonly used means for message passing in MPI is via intra-communicators. Intra-communicators contain an instance of a group, contexts of communication for both point-to-point and collective communication, and the ability to include virtual topology and other attributes. These features work as follows:

- **Contexts** provide the ability to have separate safe “universes” of message-passing in MPI. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on “other” communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator.
- **Groups** define the participants in the communication (see above) of a communicator.
6.1. INTRODUCTION

- A virtual topology defines a special mapping of the ranks in a group to and from a topology. Special constructors for communicators are defined in Chapter 7 to provide this feature. Intra-communicators as described in this chapter do not have topologies.

- Attributes define the local information that the user or library has added to a communicator for later reference.

Advice to users. The practice in many communication libraries is that there is a unique, predefined communication universe that includes all processes available when the parallel program is initiated; the processes are assigned consecutive ranks. Participants in a point-to-point communication are identified by their rank; a collective communication (such as broadcast) always involves all processes. This practice can be followed in MPI by using the predefined communicator MPI_COMM_WORLD. Users who are satisfied with this practice can plug in MPI_COMM_WORLD wherever a communicator argument is required, and can consequently disregard the rest of this chapter.

(End of advice to users.)

Inter-communicators. The discussion has dealt so far with intra-communication: communication within a group. MPI also supports inter-communication: communication between two non-overlapping groups. When an application is built by composing several parallel modules, it is convenient to allow one module to communicate with another using local ranks for addressing within the second module. This is especially convenient in a client-server computing paradigm, where either client or server are parallel. The support of inter-communication also provides a mechanism for the extension of MPI to a dynamic model where not all processes are preallocated at initialization time. In such a situation, it becomes necessary to support communication across “universes.” Inter-communication is supported by objects called inter-communicators. These objects bind two groups together with communication contexts shared by both groups. For inter-communicators, these features work as follows:

- Contexts provide the ability to have a separate safe “universe” of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on “other” communicators, and avoids the need to synchronize entry or exit into library code.

- A local and remote group specify the recipients and destinations for an inter-communicator.

- Virtual topology is undefined for an inter-communicator.

- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in an related manner to intra-communicators. Users who do not need inter-communication in their applications can safely provide the ability to have a separate safe “universe” of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on “other” communicators, and avoids the need to synchronize entry or exit into library code.

- A local and remote group specify the recipients and destinations for an inter-communicator.

- Virtual topology is undefined for an inter-communicator.

- As before, attributes cache defines the local information that the user or library has added to a communicator for later reference.

MPI provides mechanisms for creating and manipulating inter-communicators. They are used for point-to-point and collective communication in an related manner to intra-communicators. Users who do not need inter-communication in their applications can safely
ignore this extension. Users who require inter-communication between overlapping groups must layer this capability on top of MPI.

6.2 Basic Concepts

In this section, we turn to a more formal definition of the concepts introduced above.

6.2.1 Groups

A group is an ordered set of process identifiers (henceforth processes); processes are implementation-dependent objects. Each process in a group is associated with an integer rank. Ranks are contiguous and start from zero. Groups are represented by opaque group objects, and hence cannot be directly transferred from one process to another. A group is used within a communicator to describe the participants in a communication “universe” and to rank such participants (thus giving them unique names within that “universe” of communication).

There is a special pre-defined group: MPI_GROUP_EMPTY, which is a group with no members. The predefined constant MPI_GROUP_NULL is the value used for invalid group handles.

Advice to users. MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (End of advice to users.)

Advice to implementors. A group may be represented by a virtual-to-real process-address-translation table. Each communicator object (see below) would have a pointer to such a table. Simple implementations of MPI will enumerate groups, such as in a table. However, more advanced data structures make sense in order to improve scalability and memory usage with large numbers of processes. Such implementations are possible with MPI. (End of advice to implementors.)

6.2.2 Contexts

A context is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-to-point operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don’t interfere.

Advice to users. MPI_GROUP_EMPTY, which is a valid handle to an empty group, should not be confused with MPI_GROUP_NULL, which in turn is an invalid handle. The former may be used as an argument to group operations; the latter, which is returned when a group is freed, is not a valid argument. (End of advice to users.)

Advice to implementors. A group may be represented by a virtual-to-real process-address-translation table. Each communicator object (see below) would have a pointer to such a table. Simple implementations of MPI will enumerate groups, such as in a table. However, more advanced data structures make sense in order to improve scalability and memory usage with large numbers of processes. Such implementations are possible with MPI. (End of advice to implementors.)

6.2.2 Contexts

A context is a property of communicators (defined next) that allows partitioning of the communication space. A message sent in one context cannot be received in another context. Furthermore, where permitted, collective operations are independent of pending point-to-point operations. Contexts are not explicit MPI objects; they appear only as part of the realization of communicators (below).

Advice to implementors. Distinct communicators in the same process have distinct contexts. A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication. Safety means that collective and point-to-point communication within one communicator do not interfere, and that communication over distinct communicators don’t interfere.
A possible implementation for a context is as a supplemental tag attached to messages on send and matched on receive. Each intra-communicator stores the value of its two tags (one for point-to-point and one for collective communication). Communicator-generating functions use a collective communication to agree on a new group-wide unique context.

Analogously, in inter-communication, two context tags are stored per communicator, one used by group A to send and group B to receive, and a second used by group B to send and for group A to receive.

Since contexts are not explicit objects, other implementations are also possible. (End of advice to implementors.)

6.2.3 Intra-Communicators

Intra-communicators bring together the concepts of group and context. To support implementation-specific optimizations, and application topologies (defined in the next chapter, Chapter 7), communicators may also “cache” additional information (see Section 6.7).

MPI communication operations reference communicators to determine the scope and the “communication universe” in which a point-to-point or collective operation is to operate.

Each communicator contains a group of valid participants; this group always includes the local process. The source and destination of a message is identified by process rank within that group.

For collective communication, the intra-communicator specifies the set of processes that participate in the collective operation (and their order, when significant). Thus, the communicator restricts the “spatial” scope of communication, and provides machine-independent process addressing through ranks.

Intra-communicators are represented by opaque intra-communicator objects, and hence cannot be directly transferred from one process to another.

6.2.4 Predefined Intra-Communicators

An initial intra-communicator MPI_COMM_WORLD of all processes the local process can communicate with after initialization (itself included) is defined once MPI_INIT or MPI_INIT_THREAD has been called. In addition, the communicator MPI_COMM_SELF is provided, which includes only the process itself.

The predefined constant MPI_COMM_NULL is the value used for invalid communicator handles.

In a static-process-model implementation of MPI, all processes that participate in the computation are available after MPI is initialized. For this case, MPI_COMM_WORLD is a communicator of all processes available for the computation; this communicator has the same value in all processes. In an implementation of MPI where processes can dynamically join an MPI execution, it may be the case that a process starts an MPI computation without having access to all other processes. In such situations, MPI_COMM_WORLD is a communicator incorporating all processes with which the joining process can immediately communicate. Therefore, MPI_COMM_WORLD may simultaneously represent disjoint groups in different processes.

All MPI implementations are required to provide the MPI_COMM_WORLD communicator. It cannot be deallocated during the life of a process. The group corresponding to
this communicator does not appear as a pre-defined constant, but it may be accessed using
MPI_COMM_GROUP (see below). MPI does not specify the correspondence between the
process rank in MPI_COMM_WORLD and its (machine-dependent) absolute address. Neither
does MPI specify the function of the host process, if any. Other implementation-dependent,
predefined communicators may also be provided.

6.3 Group Management
This section describes the manipulation of process groups in MPI. These operations are
local and their execution does not require interprocess communication.

6.3.1 Group Accessors

MPI_GROUP_SIZE(group, size)

IN group

OUT size

number of processes in the group (integer)

int MPI_Group_size(MPI_Group group, int *size)

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)

INTEGER GROUP, SIZE, IERROR

int MPI::Group::Get_size() const

MPI_GROUP_RANK(group, rank)

IN group

OUT rank

rank of the calling process in group, or
MPI_UNDEFINED if the process is not a member (integer)

int MPI_Group_rank(MPI_Group group, int *rank)

MPI_GROUP_RANK(GROUP, RANK, IERROR)

INTEGER GROUP, RANK, IERROR

int MPI::Group::Get_rank() const
6.3. GROUP MANAGEMENT

MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)

IN group1
number of ranks in ranks1 array (MPI_Group)

IN n array of zero or more valid ranks in group1 (integer)

IN ranks1 group2 (MPI_Group)

OUT ranks2 array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists.

This function is important for determining the relative numbering of the same processes in two different groups. For instance, if one knows the ranks of certain processes in the group of MPI_COMM_WORLD, one might want to know their ranks in a subset of that group. MPI_PROC_NULL is a valid rank for input to MPI_GROUP_TRANSLATE_RANKS, which returns MPI_PROC_NULL as the translated rank.

MPI_GROUP_COMPARE(group1, group2, result)

IN group1 first group (MPI_Group)

IN group2 second group (MPI_Group)

OUT result (integer)

int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)

MPIproc_NULL results if the group members and group order is exactly the same in both groups. This happens for instance if group1 and group2 are the same handle. MPI_SIMILAR results if the group members are the same but the order is different. MPI_UNEQUAL results otherwise.

6.3.2 Group Constructors

Group constructors are used to subset and superset existing groups. These constructors construct new groups from existing groups. These are local operations, and distinct groups may be defined on different processes; a process may also define a group that does not include itself. Consistent definitions are required when groups are used as arguments in communicator-building functions. MPI does not provide a mechanism to build a group.
from scratch, but only from other, previously defined groups. The base group, upon which all other groups are defined, is the group associated with the initial communicator

```
MPI_COMM_WORLD (accessible through the function MPI_COMM_GROUP).
```

Rationale. In what follows, there is no group duplication function analogous to

```
MPI_COMM_DUP, defined later in this chapter. There is no need for a group duplica-
```
tor. A group, once created, can have several references to it by making copies of
the handle. The following constructors address the need for subsets and supersets of
existing groups. (End of rationale.)

Advice to implementors. Each group constructor behaves as if it returned a new

```
group object. When this new group is a copy of an existing group, then one can
```
avoid creating such new objects, using a reference-count mechanism. (End of advice
to implementors.)

```
MPI_COMM_GROUP(comm, group)
```

```
IN  comm  communicator (handle)
OUT group  group corresponding to comm (handle)
```

```
t int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

```
MPI_COMM_GROUP(COMM, GROUP, IERROR)
```

```
MPI::Group MPI::Comm::Get_group() const
```

```
MPI_COMM_GROUP returns in group a handle to the group of comm.
```

```
MPI_GROUP_UNION(group1, group2, newgroup)
```

```
IN  group1  first group (handle)
IN  group2  second group (handle)
OUT newgroup  union group (handle)
```

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
```

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
```

```
static MPI::Group MPI::Group::Union(const MPI::Group& group1,
```
const MPI::Group& group2)
```

```
``
6.3. GROUP MANAGEMENT

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN group1 first group (handle)
IN group2 second group (handle)
OUT newgroup intersection group (handle)

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)

INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

static MPI::Group MPI::Group::Intersect(const MPI::Group& group1, const MPI::Group& group2)

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN group1 first group (handle)
IN group2 second group (handle)
OUT newgroup difference group (handle)

int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)

INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

static MPI::Group MPI::Group::Difference(const MPI::Group& group1, const MPI::Group& group2)

The set-like operations are defined as follows:

union All elements of the first group (group1), followed by all elements of second group (group2) not in first.

intersect all elements of the first group that are also in the second group, ordered as in first group.

difference all elements of the first group that are not in the second group, ordered as in first group.

Note that for these operations the order of processes in the output group is determined primarily by order in the first group (if possible) and then, if necessary, by order in the second group. Neither union nor intersection are commutative, but both are associative.

The new group can be empty, that is, equal to MPI_GROUP_EMPTY.
1. **MPI_Group_incl**
   - **IN** `group` (handle)
   - **IN** `n` number of elements in array ranks (and size of newgroup) (integer)
   - **IN** `ranks` ranks of processes in group to appear in newgroup (array of integers)
   - **OUT** `newgroup` new group derived from above, in the order defined by ranks (handle)

   ```c
   int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
   ```

2. **MPI_Group_excl**
   - **IN** `group` (handle)
   - **IN** `n` number of elements in array ranks (and size of newgroup) (integer)
   - **IN** `ranks` ranks of processes in group to appear in newgroup (array of integers)
   - **OUT** `newgroup` new group derived from above, in the order defined by ranks (handle)

   ```c
   int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
   ```

The function `MPI_Group_incl` creates a group `newgroup` that consists of the `n` processes in `group` with ranks `rank[0]`, ... , `rank[n-1]`, the process with rank `i` in `newgroup` is the process with rank `rank[i]` in `group`. Each of the `n` elements of `ranks` must be a valid rank in `group` and all elements must be distinct, or else the program is erroneous. If `n = 0`, then `newgroup` is `MPI_Group_EMPTY`. This function can, for instance, be used to reorder the elements of a group. See also `MPI_GROUP_COMPARE`.

The function `MPI_Group_excl` creates a group of processes `newgroup` that is obtained by deleting from `group` those processes with ranks `rank[0]`, ... , `rank[n-1]`. The ordering of processes in `newgroup` is identical to the ordering in `group`. Each of the `n` elements of `ranks` must be a valid rank in `group` and all elements must be distinct; otherwise, the program is erroneous. If `n = 0`, then `newgroup` is identical to `group`. See also `MPI_GROUP_COMPARE`.
MPI_WORLD_Range_incl(group, n , ranges, newgroup)

IN group (handle)
IN n number of elements in array ranges (integer)
IN ranges a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating the ranks in group of processes to be included in newgroup
OUT newgroup new group derived from above, in the order defined by ranges (handle)

MPI_WORLD_Range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)

MPI::Group MPI::Group::Range_incl(int n, const int ranges[][3]) const

If ranges consist of the triplets

(first1, last1, stride1), ..., (firstn, lastn, striden)

then newgroup consists of the sequence of processes in group with ranks

first1, first1 + stride1, ..., first1 + \frac{last1 - first1}{stride1} \cdot stride1, ...

firstn, firstn + striden, ..., firstn + \frac{lastn - firstn}{striden} \cdot striden.

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous. Note that we may have first1 > last1, and stride1 may be negative, but cannot be zero.

The functionality of this routine is specific to being equivalent to expanding the array of ranges to an array of the included ranks and passing the resulting array of ranks and other arguments to MPI_WORLD_Range_incl. A call to MPI_WORLD_Range_incl is equivalent to a call to MPI_GROUP_RANGE_INCL with each rank i in ranks replaced by the triplet (i, i, 1) in the argument ranges.

MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)

IN group (handle)
IN n number of elements in array ranges (integer)
IN ranges a one-dimensional array of integer triplets of the form (first rank, last rank, stride) indicating the ranks in group of processes to be excluded from the output group newgroup.
OUT newgroup new group derived from above, preserving the order in group (handle)

MPI_GROUP_RANGE_EXCL(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3]) const

If ranges consist of the triplets

(first1, last1, stride1), ..., (firstn, lastn, striden)

then newgroup consists of the sequence of processes in group with ranks

first1, first1 + stride1, ..., first1 + \frac{last1 - first1}{stride1} \cdot stride1, ...

firstn, firstn + stride1, ..., firstn + \frac{lastn - firstn}{striden} \cdot striden.

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous. Note that we may have first1 > last1, and stride1 may be negative, but cannot be zero.

The functionality of this routine is specific to being equivalent to expanding the array of ranges to an array of the included ranks and passing the resulting array of ranks and other arguments to MPI_WORLD_Range_excl. A call to MPI_WORLD_Range_excl is equivalent to a call to MPI_GROUP_RANGE_EXCL with each rank i in ranks replaced by the triplet (i, i, 1) in the argument ranges.
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[3][3], MPI_Group *newgroup)

MPI::Group MPI::Group::Range_excl(int n, const int ranges[][3]) const

Each computed rank must be a valid rank in group and all computed ranks must be distinct, or else the program is erroneous.

The functionality of this routine is specified to be equivalent to expanding the array of ranges to an array of the excluded ranks and passing the resulting array of ranks and other arguments to MPI_GROUP_EXCL. A call to MPI_GROUP_EXCL is equivalent to a call to MPI::Group::Range_excl with each rank i in ranks replaced by the triplet (i,i,1) in the argument ranges.

Advice to users. The range operations do not explicitly enumerate ranks, and therefore are more scalable if implemented efficiently. Hence, we recommend MPI programmers to use them whenever possible, as high-quality implementations will take advantage of this fact. (End of advice to users.)

Advice to implementors. The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (End of advice to implementors.)

6.3.3 Group Destructors

MPI_GROUP_FREE(group)

INOUT group group (handle)

int MPI_Group_free(MPI_Group *group)

MPI::Group::Free()

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

Advice to implementors. One can keep a reference count that is incremented for each call to MPI_COMM_GROUP, MPI_COMM_CREATE and MPI_COMM_DUP, and decremented for each call to MPI_GROUP_FREE or MPI_COMM_FREE; the group object is ultimately deallocated when the reference count drops to zero. (End of advice to implementors.)
6.4 Communicator Management

This section describes the manipulation of communicators in MPI. Operations that access communicators are local and their execution does not require interprocess communication. Operations that create communicators are collective and may require interprocess communication.

Advice to implementors. High-quality implementations should amortize the overheads associated with the creation of communicators (for the same group, or subsets thereof) over several calls, by allocating multiple contexts with one collective communication. (End of advice to implementors.)

6.4.1 Communicator Accessors

The following are all local operations.

```c
MPI_Comm_size(MPI_Comm comm, int *size)
```

Rationale. This function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available (for this version of MPI, there is no standard way to change the number of processes once initialization has taken place).

This call is often used with the next call to determine the amount of concurrency available for a specific library or program. The following call, MPI_Comm_rank, indicates the rank of the process that calls it in the range from 0 . . . size−1, where size is the return value of MPI_Comm_size. (End of advice to users.)

```c
MPI_Comm_rank(MPI_Comm comm, int rank)
```

Rationale. This function is equivalent to accessing the communicator’s group with MPI_Group (see above), computing the size using MPI_GROUP_SIZE, and then freeing the temporary group via MPI_Group_free. However, this function is so commonly used, that this shortcut was introduced. (End of rationale.)

Advice to users. This function indicates the number of processes involved in a communicator. For MPI_COMM_WORLD, it indicates the total number of processes available (for this version of MPI, there is no standard way to change the number of processes once initialization has taken place).

This call is often used with the next call to determine the amount of concurrency available for a specific library or program. The following call, MPI_Comm_rank, indicates the rank of the process that calls it in the range from 0 . . . size−1, where size is the return value of MPI_Comm_size. (End of advice to users.)
Rationale. This function is equivalent to accessing the communicator’s group with `MPI_COMM_GROUP` (see above), computing the rank using `MPI_COMM_RANK`, and then freeing the temporary group via `MPI_COMM_FREE`. However, this function is so commonly used, that this shortcut was introduced. (End of rationale.)

Advice to users. This function gives the rank of the process in the particular communicator’s group. It is useful, as noted above, in conjunction with `MPI_COMM_SIZE`. Many programs will be written with the master-slave model, where one process (such as the rank-zero process) will play a supervisory role, and the other processes will serve as compute nodes. In this framework, the two preceding calls are useful for determining the roles of the various processes of a communicator. (End of advice to users.)
The MPI interface provides four communicator construction routines that apply to both intracomputers and intercommunicators. The construction routine

MPI_INTERCOMM_CREATE (discussed later) applies only to intercommunicators.

An intracommunicator involves a single group while an intercommunicator involves two groups. Where the following discussions address intercommunicator semantics, the two groups in an intracommunicator are called the left and right groups. A process in an intracommunicator is a member of either the left or the right group. From the point of view of that process, the group that the process is a member of is called the local group; the other group (relative to that process) is the remote group. The left and right group labels give us a way to describe the two groups in an intracommunicator that is not relative to any particular process (as the local and remote groups are).

```c
MPI_COMM_DUP(comm, newcomm)
```

```c
ing MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
```

```c
MPI::Intracomm MPI::Intracomm::Dup() const
MPI::Intercomm MPI::Intercomm::Dup() const
MPI::Cartcomm MPI::Cartcomm::Dup() ... MPI::Intercomm::Clone() const
MPI::Cartcomm& MPI::Cartcomm::Clone() const
MPI::Graphcomm& MPI::Graphcomm::Clone() const
```

Advice to users. This operation is used to provide a parallel library call with a duplicate communicator space that has the same properties as the original communicator. This includes any attributes (see below), and topologies (see Chapter 7). This call is valid even if there are pending point-to-point communications involving the communicator comm. A typical call might involve a MPI_COMM_DUP at the beginning of
the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the
eend of the call. Other models of communicator management are also possible.
This call applies to both intra- and inter-communicators. (End of advice to users.)

Advice to implementors. One need not actually copy the group information, but only
add a new reference and increment the reference count. Copy on write can be used
for the cached information. (End of advice to implementors.)

MPI_COMM_CREATE(comm, group, newcomm)
IN  comm communicator (handle)
IN  group Group, which is a subset of the group of
     comm (handle)
OUT newcomm new communicator (handle)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
INTEGER COMM, GROUP, NEWCOMM, IERROR
MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const
MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const
If comm is an intra-communicator, this function creates a new communicator newcomm
with communication group defined by group and a new context. No cached information
propagates from comm to newcomm. The function returns MPI_COMM_NULL to processes
that are not in group. The call is erroneous if not all group arguments have the same value,
or if group is not a subset of the group associated with comm. Note that the call is to be
executed by all processes in comm, even if they do not belong to the new group.

Rationale. The requirement that the entire group of comm participate in the call
stems from the following considerations:
• It allows the implementation to layer MPI_COMM_CREATE on top of regular
  collective communications.
• It provides additional safety, in particular in the case where partially overlapping
groups are used to create new communicators.
• It permits implementations sometimes to avoid communication related to context
  creation.
(End of rationale.)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of pro-
cesses for the purpose of separate MIMD computation, with separate communication
space. newcomm, which emerges from MPI_COMM_CREATE can be used in subse-
cquent calls to MPI_COMM_CREATE (or other communicator constructors) further to
subdivide a computation into parallel sub-computations. A more general service is
provided by MPI_COMM_SPLIT, below. (End of advice to users.)

the parallel call, and an MPI_COMM_FREE of that duplicated communicator at the
end of the call. Other models of communicator management are also possible.
This call applies to both intra- and inter-communicators. (End of advice to users.)

Advice to implementors. One need not actually copy the group information, but only
add a new reference and increment the reference count. Copy on write can be used
for the cached information. (End of advice to implementors.)

MPI_COMM_CREATE(comm, group, newcomm)
IN  comm communicator (handle)
IN  group Group, which is a subset of the group of
     comm (handle)
OUT newcomm new communicator (handle)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
INTEGER COMM, GROUP, NEWCOMM, IERROR
MPI::Intercomm MPI::Intercomm::Create(const MPI::Group& group) const
MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& group) const
If comm is an intra-communicator, this function creates a new communicator newcomm
with communication group defined by group and a new context. No cached information
propagates from comm to newcomm. The function returns MPI_COMM_NULL to processes
that are not in group. The call is erroneous if not all group arguments have the same value,
or if group is not a subset of the group associated with comm. Note that the call is to be
executed by all processes in comm, even if they do not belong to the new group.

Rationale. The requirement that the entire group of comm participate in the call
stems from the following considerations:
• It allows the implementation to layer MPI_COMM_CREATE on top of regular
  collective communications.
• It provides additional safety, in particular in the case where partially overlapping
groups are used to create new communicators.
• It permits implementations sometimes to avoid communication related to context
  creation.
(End of rationale.)

Advice to users. MPI_COMM_CREATE provides a means to subset a group of pro-
cesses for the purpose of separate MIMD computation, with separate communication
space. newcomm, which emerges from MPI_COMM_CREATE can be used in subse-
cquent calls to MPI_COMM_CREATE (or other communicator constructors) further to
subdivide a computation into parallel sub-computations. A more general service is
provided by MPI_COMM_SPLIT, below. (End of advice to users.)
Advice to implementors. Since all processes calling MPI_COMM_DUP or MPI_COMM_CREATE provide the same group argument, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system should be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (End of advice to implementors.)

If comm is an intercommunicator, then the output communicator is also an intercommunicator where the local group consists only of those processes contained in group (see Figure 6.1). The group argument should only contain those processes in the local group of the input intercommunicator that are to be a part of newcomm. If either group does not specify at least one process in the local group of the intercommunicator, or if the calling process is not included in the group, MPI_COMM_NULL is returned.

Rationale. In the case where either the left or right group is empty, a null communicator is returned instead of an intercommunicator with MPI_GROUP_EMPTY because the side with the empty group must return MPI_COMM_NULL. (End of rationale.)

Figure 6.1: Intercommunicator create using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.
Example 6.1 The following example illustrates how the first node in the left side of an intercommunicator could be joined with all members on the right side of an intercommunicator to form a new intercommunicator.

```c
MPI_Comm inter_comm, new_inter_comm;
MPI_Group local_group, group;
int rank = 0; /* rank on left side to include in new inter-comm */
/* Construct the original intercommunicator: "inter_comm" */
...
/* Construct the group of processes to be in new intercommunicator */
if (/* I'm on the left side of the intercommunicator */) {
    MPI_Comm_group ( inter_comm, &local_group );
    MPI_Group_incl ( local_group, 1, &rank, &group );
    MPI_Group_free ( &local_group );
} else
    MPI_Comm_group ( inter_comm, &group );
MPI_Comm_create ( inter_comm, group, &new_inter_comm );
MPI_Group_free( &group );
```

This function partitions the group associated with `comm` into disjoint subgroups, one for each value of `color`. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument `key`, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in `newcomm`. A process may supply the color value `MPI_UNDEFINED`, in which case `newcomm` returns `MPI_COMM_NULL`. This is a collective call, but each process is permitted to provide different values for `color` and `key`.

```c
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
IN comm communicator (handle)
IN color control of subset assignment (integer)
IN key control of rank assignment (integer)
OUT newcomm new communicator (handle)
```

This function partitions the group associated with `comm` into disjoint subgroups, one for each value of `color`. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument `key`, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in `newcomm`. A process may supply the color value `MPI_UNDEFINED`, in which case `newcomm` returns `MPI_COMM_NULL`. This is a collective call, but each process is permitted to provide different values for `color` and `key`.
A call to `MPI_COMM_CREATE(comm, group, newcomm)` is equivalent to a call to `MPI_COMM_SPLIT(comm, color, key, newcomm)`, where all members of `group` provide `color = 0` and `key = rank in group`, and all processes that are not members of `group` provide `color = MPI_UNDEFINED`. The function `MPI_COMM_SPLIT` allows more general partitioning of a group into one or more subgroups with optional reordering.

The value of `color` must be nonnegative.

**Advice to users.** This is an extremely powerful mechanism for dividing a single communicating group of processes into \(k\) subgroups, with \(k\) chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra.

Multiple calls to `MPI_COMM_SPLIT` can be used to overcome the requirement that any call have no overlap of the resulting communicators (each process is of only one color per call). In this way, multiple overlapping communication structures can be created. Creative use of the `color` and `key` in such splitting operations is encouraged.

Note that, for a fixed color, the keys need not be unique. It is `MPI_COMM_SPLIT`'s responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group.

Essentially, making the key value zero for all processes of a given color means that one doesn’t really care about the rank-order of the processes in the new communicator.

(End of advice to users.)

**Rationale.** `color` is restricted to be nonnegative, so as not to conflict with the value assigned to `MPI_UNDEFINED`. (End of rationale.)

The result of `MPI_COMM_SPLIT` on an intercommunicator is that those processes on the left with the same `color` as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 6.2). For those colors that are specified only on one side of the intercommunicator, `MPI_COMM_NULL` is returned. `MPI_COMM_NULL` is also returned to those processes that specify `MPI_UNDEFINED` as the color.

**Example 6.2** (Parallel client-server model). The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```c
/* Client code */
MPI_Comm multiple_server_comm;
MPI_Comm single_server_comm;
int color, rank, num_servers;

/* Create intercommunicator with clients and servers: */
multiple_server_comm *=
```

(Continued on page 199)

---

A call to `MPI_COMM_CREATE(comm, group, newcomm)` is equivalent to a call to `MPI_COMM_SPLIT(comm, color, key, newcomm)`, where all members of `group` provide `color = 0` and `key = rank in group`, and all processes that are not members of `group` provide `color = MPI_UNDEFINED`. The function `MPI_COMM_SPLIT` allows more general partitioning of a group into one or more subgroups with optional reordering.

The value of `color` must be nonnegative.

**Advice to users.** This is an extremely powerful mechanism for dividing a single communicating group of processes into \(k\) subgroups, with \(k\) chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra.

Multiple calls to `MPI_COMM_SPLIT` can be used to overcome the requirement that any call have no overlap of the resulting communicators (each process is of only one color per call). In this way, multiple overlapping communication structures can be created. Creative use of the `color` and `key` in such splitting operations is encouraged.

Note that, for a fixed color, the keys need not be unique. It is `MPI_COMM_SPLIT`'s responsibility to sort processes in ascending order according to this key, and to break ties in a consistent way. If all the keys are specified in the same way, then all the processes in a given color will have the relative rank order as they did in their parent group.

Essentially, making the key value zero for all processes of a given color means that one doesn’t really care about the rank-order of the processes in the new communicator.

(End of advice to users.)

**Rationale.** `color` is restricted to be nonnegative, so as not to conflict with the value assigned to `MPI_UNDEFINED`. (End of rationale.)

The result of `MPI_COMM_SPLIT` on an intercommunicator is that those processes on the left with the same `color` as those processes on the right combine to create a new intercommunicator. The key argument describes the relative rank of processes on each side of the intercommunicator (see Figure 6.2). For those colors that are specified only on one side of the intercommunicator, `MPI_COMM_NULL` is returned. `MPI_COMM_NULL` is also returned to those processes that specify `MPI_UNDEFINED` as the color.

**Example 6.2** (Parallel client-server model). The following client code illustrates how clients on the left side of an intercommunicator could be assigned to a single server from a pool of servers on the right side of an intercommunicator.

```c
/* Client code */
MPI_Comm multiple_server_comm;
MPI_Comm single_server_comm;
int color, rank, num_servers;

/* Create intercommunicator with clients and servers: */
multiple_server_comm *=
```
Figure 6.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT extended to intercommunicators.
/* Find out the number of servers available */
MPI_Comm_remote_size ( multiple_server_comm, &num_servers );

/* Determine my color */
MPI_Comm_rank ( multiple_server_comm, &rank );
color = rank % num_servers;

/* Split the intercommunicator */
MPI_Comm_split ( multiple_server_comm, color, rank,
                  &single_server_comm );

The following is the corresponding server code:

/* Server code */
MPI_Comm multiple_client_comm; MPI_Comm single_server_comm; int rank;

/* Create intercommunicator with clients and servers: */
multiple_client_comm */
...

/* Split the intercommunicator for a single server per group of clients */
MPI_Comm_rank ( multiple_client_comm, &rank );
MPI_Comm_split ( multiple_client_comm, rank, 0,
                  &single_server_comm );

6.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

INOUT comm communicator to be destroyed (handle)

int MPI_Comm_free(MPI_Comm *comm)

MPI_COMM_FREE(comm, ierr)

INTEGER comm, ierr

void MPI::Comm::Free()

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 6.7) are called in arbitrary order.

6.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

INOUT comm communicator to be destroyed (handle)

int MPI_Comm_free(MPI_Comm *comm)

MPI_COMM_FREE(comm, ierr)

INTEGER comm, ierr

void MPI::Comm::Free()

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 6.7) are called in arbitrary order.
Advice to implementors. A reference-count mechanism may be used: the reference count is incremented by each call to MPI_COMM_DUP, and decremented by each call to MPI_COMM_FREE. The object is ultimately deallocated when the count reaches zero.

Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (End of advice to implementors.)

6.5 Motivating Examples

6.5.1 Current Practice #1

Example #1a:

```c
main(int argc, char **argv)
{
    int me, size;
    ...
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    (void)printf("Process %d size %d\n", me, size);
    ...
    MPI_Finalize();
}
```

Example #1b (supposing that size is even):

```c
main(int argc, char **argv)
{
    int me, size;
    int SOME_TAG = 0;
    ...
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
    MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */
    if((me % 2) == 0)
    {
        /* send unless highest-numbered process */
        if((me + 1) < size)
            MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
    }
}
```
Example #1b schematically illustrates message exchanges between “even” and “odd” processes in the “all” communicator.

6.5.2 Current Practice #2

```c
main(int argc, char **argv)
{
    int me, count;
    void *data;
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (me == 0) {
        /* get input, create buffer ''data'' */
        ...
    } else {
        MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
        ...
    }
    MPI_Finalize();
}
```

This example illustrates the use of a collective communication.

6.5.3 (Approximate) Current Practice #3

```c
main(int argc, char **argv)
{
    int me, count, count2;
    void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
    MPI_Group MPI_GROUP_WORLD, grprem;
    MPI_Comm(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
    MPI_Comm_rank(MPI_COMM_WORLD, &me); /* local */
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (me == 0) {
        /* get input, create buffer ''data'' */
        ...
    } else {
        MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
        ...
    }
    MPI_Finalize();
}
```

This example illustrates the use of a collective communication.
MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem); /* local */
MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

if(me != 0) {
    /* compute on slave */
    ...
    MPI_Reduce(send_buf, recv_buff, count, MPI_INT, MPI_SUM, 1, commslave);
    ...
    MPI_Comm_free(&commslave);
}

MPI_Reduce(send_buf2, recv_buff2, count2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Group_free(&MPI_GROUP_WORLD);
MPI_Group_free(&grprem);
MPI_Finalize();

This example illustrates how a group consisting of all but the zeroth process of the “all”
group is created, and then how a communicator is formed (commslave) for that new group.
The new communicator is used in a collective call, and all processes execute a collective call
in the MPI_COMM_WORLD context. This example illustrates how the two communicators
(that inherently possess distinct contexts) protect communication. That is, communication
in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.
In summary, “group safety” is achieved via communicators because distinct contexts
within communicators are enforced to be unique on any process.

6.5.4 Example #4

The following example is meant to illustrate “safety” between point-to-point and collective
communication. MPI guarantees that a single communicator can do safe point-to-point and
collective communication.

#define TAG_ARBITRARY 12345
#define SOME_COUNT 50

main(int argc, char **argv) {
    int me;
    MPI_Request request[2];
    MPI_Status status[2];
    MPI_Group MPI_GROUP_WORLD, subgroup;
    int ranks[] = { 2, 4, 6, 8 };  
    MPI_Comm the_comm;
    ...
    MPI_Init(&argc, &argv);
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
MPI_Group_rank(subgroup, &me); /* local */
MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
if(me != MPI_UNDEFINED)
{
  MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
    the_comm, request);
  MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
    the_comm, request+1);
  for(i = 0; i < SOME_COUNT, i++)
    MPI_Reduce(..., the_comm);
  MPI_Waitall(2, request, status);
}
MPI_Comm_free(&the_comm);
MPI_Group_free(&MPI_GROUP_WORLD);
MPI_Group_free(&subgroup);
MPI_Finalize();

6.5.5 Library Example #1

The main program:

main(int argc, char **argv)
{
  int done = 0;
  user_lib_t *libh_a, *libh_b;
  void *dataset1, *dataset2;
  ...
  MPI_Init(&argc, &argv);
  ...
  init_user_lib(MPI_COMM_WORLD, &libh_a);
  init_user_lib(MPI_COMM_WORLD, &libh_b);
  ...
  user_start_op(libh_a, dataset1);
  user_start_op(libh_b, dataset2);
  ...
  while(!done)
  {
    /* work */
    ...
    MPI_Reduce(..., MPI_COMM_WORLD);
    ...
  
  ...
  MPI_Finalize();
}
The user library initialization code:

```c
void init_user_lib(MPI_Comm comm, user_lib_t **handle)
{
    user_lib_t *save; user_lib_initsave(&save); /* local */
    MPI_Comm_dup(comm, &(save -> comm)); /* other inits */
    *handle = save;
}
```

User start-up code:

```c
void user_start_op(user_lib_t *handle, void *data)
{
    MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
    MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
}
```

User communication clean-up code:

```c
void user_end_op(user_lib_t *handle)
{
    MPI_Status status;
    MPI_Wait(handle -> isend_handle, &status);
    MPI_Wait(handle -> irecv_handle, &status);
}
```

User object clean-up code:

```c
void uninit_user_lib(user_lib_t *handle)
{
    MPI_Comm_free(&(handle -> comm));
    free(handle);
}
```
6.5. MOTIVATING EXAMPLES

6.5.6 Library Example #2

The main program:

```c
main(int argc, char **argv) {
    int ma, mb;
    MPI_Group MPI_GROUP_WORLD, group_a, group_b;
    MPI_Comm comm_a, comm_b;

    static int list_a[] = {0, 1};
    #if defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
    static int list_b[] = {0, 2, 3};
    #else/* EXAMPLE_2A */
    static int list_b[] = {0, 2};
    #endif

    int size_list_a = sizeof(list_a)/sizeof(int);
    int size_list_b = sizeof(list_b)/sizeof(int);
    ...

    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
    MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
    MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);
    MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
    MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

    if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
    if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);

    if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);
    if(comm_b != MPI_COMM_NULL)
        lib_call(comm_b);

    if(comm_a != MPI_COMM_NULL)
        MPI_Comm_free(&comm_a);
    if(comm_b != MPI_COMM_NULL)
        MPI_Comm_free(&comm_b);
    MPI_Group_free(&group_a);
    MPI_Group_free(&group_b);
}
```
The library:

```c
void lib_call(MPI_Comm comm) {
    int me, done = 0;
    MPI_Status status;
    MPI_Comm_rank(comm, &me);
    if (me == 0) {
        while (!done) {
            MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
            /* work */
        }
    } else {
        /* work */
    }
}
#endif
```

The above example is really three examples, depending on whether or not one includes rank 3 in `lib_call`, and whether or not a synchronize is included in `lib_call`. This example illustrates that, despite contexts, subsequent calls to `lib_call` with the same context need not be safe from one another (colloquially, “back-masking”). Safety is realized if the MPI_BARRIER is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back masking.

Algorithms like “reduce” and “allreduce” have strong enough source selectivity properties so that they are inherently okay (no backmasking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [45]). Here we rely on two guarantees of MPI: pairwise ordering of messages between processes in the same context, and source selectivity — deleting either feature removes the guarantee that backmasking cannot be required.

Algorithms that try to do non-deterministic broadcasts or other calls that include wildcard operations will not generally have the good properties of the deterministic implementations of “reduce,” “allreduce,” and “broadcast.” Such algorithms would have to utilize the monotonically increasing tags (within a communicator scope) to keep things straight.

All of the foregoing is a supposition of “collective calls” implemented with point-to-point operations. MPI implementations may or may not implement collective calls using point-to-point operations. These algorithms are used to illustrate the issues of correctness and safety, independent of how MPI implements its collective calls. See also Section 6.9.
6.6 Inter-Communication

This section introduces the concept of inter-communication and describes the portions of MPI that support it. It describes support for writing programs that contain user-level servers.

All communication described thus far has involved communication between processes that are members of the same group. This type of communication is called “intra-communication” and the communicator used is called an “intra-communicator,” as we have noted earlier in the chapter.

In modular and multi-disciplinary applications, different process groups execute distinct modules and processes within different modules communicate with one another in a pipeline or a more general module graph. In these applications, the most natural way for a process to specify a target process is by the rank of the target process within the target group. In applications that contain internal user-level servers, each server may be a process group that provides services to one or more clients, and each client may be a process group that uses the services of one or more servers. It is again most natural to specify the target process by rank within the target group in these applications. This type of communication is called “inter-communication” and the communicator used is called an “inter-communicator,” as introduced earlier.

An inter-communication is a point-to-point communication between processes in different groups. The group containing a process that initiates an inter-communication operation is called the “local group,” that is, the sender in a send and the receiver in a receive. The group containing the target process is called the “remote group,” that is, the receiver in a send and the sender in a receive. As in intra-communication, the target process is specified using a (communicator, rank) pair. Unlike intra-communication, the rank is relative to a second, remote group.

All inter-communicator constructors are blocking and require that the local and remote groups be disjoint.

**Advice to users.** The groups must be disjoint for several reasons. Primarily, this is the intent of the intercommunicators — to provide a communicator for communication between disjoint groups. This is reflected in the definition of MPI_INTERCOMM_MERGE, which allows the user to control the ranking of the processes in the created intracommmunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to intercommunicators makes the most sense when the groups are disjoint. (End of advice to users.)

Here is a summary of the properties of inter-communication and inter-communicators:

- The syntax of point-to-point and collective communication is the same for both inter- and intra-communication. The same communicator can be used both for send and for receive operations.
- A target process is addressed by its rank in the remote group, both for sends and for receives.
- Communications using an inter-communicator are guaranteed not to conflict with any communications that use a different communicator.
- A communicator will provide either intra- or inter-communication, never both.
The routine MPI_COMM_TEST_INTER may be used to determine if a communicator is an inter- or intra-communicator. Inter-communicators can be used as arguments to some of the other communicator access routines. Inter-communicators cannot be used as input to some of the constructor routines for intra-communicators (for instance, MPI_COMM_CREATE).

Advice to implementors. For the purpose of point-to-point communication, communicators can be represented in each process by a tuple consisting of:

- group
- send_context
- receive_context
- source

For inter-communicators, group describes the remote group, and source is the rank of the process in the local group. For intra-communicators, group is the communicator group (remote=local), source is the rank of the process in this group, and send context and receive context are identical. A group can be represented by a rank-to-absolute-address translation table.

The inter-communicator cannot be discussed sensibly without considering processes in both the local and remote groups. Imagine a process $P$ in group $P$, and a process $Q$ in group $Q$, which has an inter-communicator $C_{PQ}$. Then

- $C_{PQ}.group$ describes the group $Q$ and $C_{PQ}.group$ describes the group $P$.
- $C_{PQ}.send.context = C_{PQ.receive.context}$ and the context is unique in $Q$.
- $C_{PQ.receive.context} = C_{PQ.send.context}$ and this context is unique in $P$.
- $C_{PQ}.source$ is rank of $P$ in $P$ and $C_{PQ}.source$ is rank of $Q$ in $Q$.

Assume that $P$ sends a message to $Q$ using the inter-communicator. Then $P$ uses the group table to find the absolute address of $Q$; source and send context are appended to the message.

Assume that $Q$ posts a receive with an explicit source argument using the inter-communicator. Then $Q$ matches receive context to the message context and source argument to the message source.

The same algorithm is appropriate for intra-communicators as well.

In order to support inter-communicator accessors and constructors, it is necessary to supplement this model with additional structures, that store information about the local communication group, and additional safe contexts. (End of advice to implementors.)

### 6.6.1 Inter-communicator Accessors

**MPI_COMM_TEST_INTER**

```
IN comm communicator (handle)
OUT flag (logical)
```
This local routine allows the calling process to determine if a communicator is an inter-communicator or an intra-communicator. It returns true if it is an inter-communicator, otherwise false.

When an inter-communicator is used as an input argument to the communicator accessors described above under intra-communication, the following table describes behavior.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_COMM_SIZE</td>
<td>returns the size of the local group.</td>
</tr>
<tr>
<td>MPI_COMM_GROUP</td>
<td>returns the local group.</td>
</tr>
<tr>
<td>MPI_COMM_RANK</td>
<td>returns the rank in the local group</td>
</tr>
</tbody>
</table>

Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)

Furthermore, the operation MPI_COMM_COMPARE is valid for inter-communicators. Both communicators must be either intra- or inter-communicators, or else MPI_UNEQUAL results. Both corresponding local and remote groups must compare correctly to get the results MPI_CONGRUENT and MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an inter-communicator:

The following are all local operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_COMM_REMOTE_SIZE(comm, size)</td>
<td>returns the number of processes in the remote group of comm</td>
</tr>
<tr>
<td>MPI_COMM_REMOTE_GROUP(comm, group)</td>
<td>returns the remote group corresponding to comm</td>
</tr>
</tbody>
</table>

Table 6.1: MPI_COMM_* Function Behavior (in Inter-Communication Mode)
6.6.2 Inter-communicator Operations

This section introduces four blocking inter-communicator operations.

MPI_INTERCOMM_CREATE is used to bind two intra-communicators into an inter-communicator; the function MPI_INTERCOMM_MERGE creates an intra-communicator by merging the local and remote groups of an inter-communicator. The functions MPI_COMM_DUP and MPI_COMM_FREE, introduced previously, duplicate and free an inter-communicator, respectively.

Overlap of local and remote groups that are bound into an inter-communicator is prohibited. If there is overlap, then the program is erroneous and is likely to deadlock. (If a process is multithreaded, and MPI calls block only a thread, rather than a process, then “dual membership” can be supported. It is then the user’s responsibility to make sure that calls on behalf of the two “roles” of a process are executed by two independent threads.)

The function MPI_INTERCOMM_CREATE can be used to create an inter-communicator from two existing intra-communicators, in the following situation: At least one selected member from each group (the “group leader”) has the ability to communicate with the selected member from the other group; that is, a “peer” communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator.

Furthermore, members of each group know the rank of their leader.

Construction of an inter-communicator from two intra-communicators requires separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.

In standard MPI implementations (with static process allocation at initialization), the MPI_COMM_WORLD communicator (or preferably a dedicated duplicate thereof) can be this peer communicator. For applications that have used spawn or join, it may be necessary to first create an intracommunicator to be used as peer.

The application topology functions described in Chapter 7 do not apply to inter-communicators. Users that require this capability should utilize MPI_INTERCOMM_MERGE to build an intra-communicator, then apply the graph or cartesian topology capabilities to that intra-communicator, creating an appropriate topology-oriented intra-communicator. Alternatively, it may be reasonable to devise one’s own application topology mechanisms for this case, without loss of generality.
6.6. INTER-COMMUNICATION

MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm)

IN  local_comm  local intra-communicator (handle)
IN  local_leader  rank of local group leader in local_comm (integer)
IN  peer_comm  "peer" communicator; significant only at the local_leader (handle)
IN  remote_leader  rank of remote group leader in peer_comm; significant only at the local_leader (integer)
IN  tag  "safe" tag (integer)
OUT newintercomm  new inter-communicator (handle)

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
MPI_Comm peer_comm, int remote_leader, int tag,
MPI_Comm *newintercomm)

MPI_INTERCOMM_CREATE(LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER,
TAG, NEWINTERCOMM, IERROR)

INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
NEWINTERCOMM, IERROR

MPI::Intercomm MPI::Intracomm::Create_intercomm(int local_leader, const
MPI::Comm peer_comm, int remote_leader, int tag) const

This call creates an inter-communicator. It is collective over the union of the local and remote groups. Processes should provide identical local_comm and local_leader arguments within each group. Wildcards are not permitted for remote_leader, local_leader, and tag.

This call uses point-to-point communication with communicator peer_comm, and with tag tag between the leaders. Thus, care must be taken that there be no pending communication on peer_comm that could interfere with this communication.

Advice to users. We recommend using a dedicated peer communicator, such as a duplicate of MPI_COMM_WORLD, to avoid trouble with peer communicators. (End of advice to users.)

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

IN  intercomm  Inter-Communicator (handle)
IN  high  (logical)
OUT newintracomm  new intra-communicator (handle)

int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
MPI_Comm *newintracomm)

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, INTRACOMM, IERROR)

INTEGER INTERCOMM, INTRACOMM, IERROR

LOGICAL HIGH
This function creates an intra-communicator from the union of the two groups that are associated with `intercomm`. All processes should provide the same high value within each of the two groups. If processes in one group provided the value `high = false` and processes in the other group provided the value `high = true` before the "high" group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

### Advice to implementors

The implementation of `MPI_INTERCOMM_MERGE`, `MPI_COMM_FREE` and `MPI_COMM_DUP` are similar to the implementation of `MPI_INTERCOMM_CREATE`, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (End of advice to implementors.)

### 6.6.3 Inter-Communication Examples

#### Example 1: Three-Group "Pipeline"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

```c
main(int argc, char **argv)
{
    MPI_Comm myComm; /* intra-communicator of local sub-group */
    MPI_Comm myFirstComm; /* inter-communicator */
    MPI_Comm mySecondComm; /* second inter-communicator (group 1 only) */
    int membershipKey;
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    /* User code must generate membershipKey in the range [0, 1, 2] */
    membershipKey = rank % 3;
}
```
/* Build intra-communicator for local sub-group */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

/* Build inter-communicators. Tags are hard-coded. */
if (membershipKey == 0) {
    /* Group 0 communicates with group 1. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1, 1, &myFirstComm);
} else if (membershipKey == 1) {
    /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0, 1, &myFirstComm);
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2, 12, &mySecondComm);
} else if (membershipKey == 2) {
    /* Group 2 communicates with group 1. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1, 12, &myFirstComm);
}

/* Do work ... */
switch(membershipKey) /* free communicators appropriately */{
    case 1:
        MPI_Comm_free(&mySecondComm);
        break;
    case 0:
        break;
    case 2:
        MPI_Comm_free(&myFirstComm);
        break;
}

MPI_Finalize();

/* Build intra-communicator for local sub-group */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

/* Build inter-communicators. Tags are hard-coded. */
if (membershipKey == 0) {
    /* Group 0 communicates with group 1. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1, 1, &myFirstComm);
} else if (membershipKey == 1) {
    /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0, 1, &myFirstComm);
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 2, 12, &mySecondComm);
} else if (membershipKey == 2) {
    /* Group 2 communicates with group 1. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1, 12, &myFirstComm);
}

/* Do work ... */
switch(membershipKey) /* free communicators appropriately */{
    case 1:
        MPI_Comm_free(&mySecondComm);
        break;
    case 0:
        break;
    case 2:
        MPI_Comm_free(&myFirstComm);
        break;
}

MPI_Finalize();
Example 2: Three-Group “Ring”
Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate.
Therefore, each requires two inter-communicators.

```c
main(int argc, char **argv)
{
    MPI_Comm myComm; /* intra-communicator of local sub-group */
    MPI_Comm myFirstComm; /* inter-communicators */
    MPI_Comm mySecondComm;
    MPI_Status status;
    int membershipKey;
    int rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ... /* User code must generate membershipKey in the range [0, 1, 2] */
    membershipKey = rank % 3;
    /* Build intra-communicator for local sub-group */
    MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
    /* Build inter-communicators. Tags are hard-coded. */
    if (membershipKey == 0) /* Group 0 communicates with groups 1 and 2. */
        { /* Group 0 communicates with groups 1 and 2. */
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                1, &myFirstComm);
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                2, &mySecondComm);
        }
    else if (membershipKey == 1) /* Group 1 communicates with groups 0 and 2. */
        { /* Group 1 communicates with groups 0 and 2. */
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                1, &myFirstComm);
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                12, &mySecondComm);
        }
    else if (membershipKey == 2) /* Group 2 communicates with groups 0 and 1. */
        { /* Group 2 communicates with groups 0 and 1. */
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                12, &myFirstComm);
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                12, &mySecondComm);
        }
    /* Do some work ... */
}
```

Example 2: Three-Group “Ring”
Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate.
Therefore, each requires two inter-communicators.

```c
main(int argc, char **argv)
{
    MPI_Comm myComm; /* intra-communicator of local sub-group */
    MPI_Comm myFirstComm; /* inter-communicators */
    MPI_Comm mySecondComm;
    MPI_Status status;
    int membershipKey;
    int rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ... /* User code must generate membershipKey in the range [0, 1, 2] */
    membershipKey = rank % 3;
    /* Build intra-communicator for local sub-group */
    MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);
    /* Build inter-communicators. Tags are hard-coded. */
    if (membershipKey == 0) /* Group 0 communicates with groups 1 and 2. */
        { /* Group 0 communicates with groups 1 and 2. */
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                1, &myFirstComm);
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                12, &mySecondComm);
        }
    else if (membershipKey == 1) /* Group 1 communicates with groups 0 and 2. */
        { /* Group 1 communicates with groups 0 and 2. */
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                1, &myFirstComm);
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                12, &mySecondComm);
        }
    else if (membershipKey == 2) /* Group 2 communicates with groups 0 and 1. */
        { /* Group 2 communicates with groups 0 and 1. */
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                12, &myFirstComm);
            MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                12, &mySecondComm);
        }
    /* Do some work ... */
}```
Example 3: Building Name Service for Intercommunication

The following procedures exemplify the process by which a user could create name service for building intercommunicators via a rendezvous involving a server communicator, and a tag name selected by both groups.

After all MPI processes execute MPI_INIT, every process calls the example function, Init_server(), defined below. Then, if the new_world returned is NULL, the process getting NULL is required to implement a server function, in a reactive loop, Do_server(). Everyone else just does their prescribed computation, using new_world as the new effective "global" communicator. One designated process calls Undo_Server() to get rid of the server when it is not needed any longer.

Features of this approach include:

- Support for multiple name servers
- Ability to scope the name servers to specific processes
- Ability to make such servers come and go as desired.

#define INIT_SERVER_TAG_1 666#define UNDO_SERVER_TAG_1 777static int server_key_val; /* for attribute management for server_comm, copy callback: */

void handle_copy_fn(MPI_Comm *oldcomm, int *keyval, void *extra_state, 
void *attribute_val_in, void **attribute_val_out, int *flag) 
{
/* copy the handle */
*attribute_val_out = attribute_val_in;
*flag = 1; /* indicate that copy to happen */
}

int Init_server(peer_comm, rank_of_server, server_comm, new_world) 
MPI_Comm peer_comm;
int rank_of_server;
MPI_Comm *server_comm; 
MPI_Comm *new_world; /* new effective world, sans server */
{
MPI_Comm temp_comm, lone_comm;
MPI_Group peer_group, temp_group;
int rank_in_peer_comm, size, color, key = 0;
/* Then free communicators before terminating... */
MPI_Comm_free(&myFirstComm);
MPI_Comm_free(&mySecondComm);
MPI_Comm_free(&myComm);
MPI_Finalize();
}
int peer_leader, peer_leader_rank_in_temp_comm;
MPI_Comm_rank(peer_comm, &rank_in_peer_comm);
MPI_Comm_size(peer_comm, &size);
if ((size < 2) || (0 > rank_of_server) || (rank_of_server >= size))
    return (MPI_ERR_OTHER);

/* create two communicators, by splitting peer_comm into the server process, and everyone else */
peer_leader = (rank_of_server + 1) % size; /* arbitrary choice */
if ((color = (rank_in_peer_comm == rank_of_server))
    {
        MPI_Comm_split(peer_comm, color, key, &lone_comm);
        MPI_Intercomm_create(lone_comm, 0, peer_comm, peer_leader,
            INIT_SERVER_TAG_1, server_comm);
        MPI_Comm_free(&lone_comm);
        *new_world = MPI_COMM_NULL;
    }
else{
        MPI_Comm_split(peer_comm, color, key, &temp_comm);
        MPI_Comm_group(peer_comm, &peer_group);
        MPI_Comm_group(temp_comm, &temp_group);
        MPI_Group_translate_ranks(peer_group, 1, &peer_leader,
            temp_group, &peer_leader_rank_in_temp_comm);
        MPI_Intercomm_create(temp_comm, peer_leader_rank_in_temp_comm,
            peer_comm, rank_of_server,
            INIT_SERVER_TAG_1, server_comm);
        /* attach new_world communication attribute to server_comm: */
        /* CRITICAL SECTION FOR MULTITHREADING */
        if(server_keyval == MPI_KEYVAL_INVALID)
            {
                /* acquire the process-local name for the server keyval */
                MPI_keyval_create(handle_copy_fn, NULL,
                    &server_keyval, NULL);
            }
        *new_world = temp_comm;
    }
/* Cache handle of intra-communicator on inter-communicator: */
MPI_Attr_put(server_comm, server_keyval, (void *)(*new_world));
}
return (MPI_SUCCESS);
}

The actual server process would commit to running the following code:

```c
int Do_server(server_comm)MPI_Comm server_comm;
{
    void init_queue();
    int en_queue(), de_queue(); /* keep triplets of integers 
    for later matching (fns not shown) */

    MPI_Comm comm;
    MPI_Status status;
    int client_tag, client_source;
    int client_rank_in_new_world, pairs_rank_in_new_world;
    int buffer[10], count = 1;
    void *queue;
    init_queue(&queue);

    for (;;)
    {
        MPI_Recv(buffer, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, 
        server_comm, &status); /* accept from any client */

        /* determine client: */
        client_tag = status.MPI_TAG;
        client_source = status.MPI_SOURCE;
        client_rank_in_new_world = buffer[0];

        if (client_tag == UNDO_SERVER_TAG_1) /* client that 
            terminates server */
        {
            while (de_queue(queue, MPI_ANY_TAG, &pairs_rank_in_new_world, 
            &pairs_rank_in_server))
            {
                MPI_Comm_free(&server_comm);
                break;
            }

            if (de_queue(queue, client_tag, &pairs_rank_in_new_world,
            &pairs_rank_in_server))
            {
                MPI_Comm_free(&server_comm);
                break;
            }
        }
    }
}
```
&pairs_rank_in_server))
{
    /* matched pair with same tag, tell them
     about each other! */
    buffer[0] = pairs_rank_in_new_world;
    MPI_Send(buffer, 1, MPI_INT, client_src, client_tag,
              server_comm);
    buffer[0] = client_rank_in_new_world;
    MPI_Send(buffer, 1, MPI_INT, pairs_rank_in_server, client_tag,
              server_comm);
}
else
    en_queue(queue, client_tag, client_source,
             client_rank_in_new_world);
}

A particular process would be responsible for ending the server when it is no longer
needed. Its call to Undo_server would terminate server function.

int Undo_server(server_comm) /* example client that ends server */
MPI_Comm *server_comm;
{
    int buffer = 0;
    MPI_Send(&buffer, 1, MPI_INT, 0, UNDO_SERVER_TAG_1, *server_comm);
    MPI_Comm_free(server_comm);
}

The following is a blocking name-service for inter-communication, with same semantic
restrictions as MPI_Intercomm_create, but simplified syntax. It uses the functionality just
defined to create the name service.

int Intercomm_name_create(local_comm, server_comm, tag, comm)
MPI_Comm local_comm, server_comm; int tag;
MPI_Comm *comm;
{
    int error;
    int found; /* attribute acquisition mgmt for new_world */
    /* comm in server_comm */
    void *val;
    MPI_Comm new_world;
    int buffer[10], rank;
    int local_leader = 0;
6.7. CACHING

MPI provides a “caching” facility that allows an application to attach arbitrary pieces of information, called attributes, to three kinds of MPI objects, communicators, windows and datatypes. More precisely, the caching facility allows a portable library to do the following:

• pass information between calls by associating it with an MPI intra- or inter-communicator, window or datatype,
• quickly retrieve that information, and
• be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

The caching capabilities, in some form, are required by built-in MPI routines such as collective communication and application topology. Defining an interface to these capabilities as part of the MPI standard is valuable because it permits routines like collective communication and application topologies to be implemented as portable code, and also because it makes MPI more extensible by allowing user-written routines to use standard MPI calling sequences.

Advice to users. The communicator MPI_COMM_SELF is a suitable choice for posting process-local attributes, via this attributing-caching mechanism. (End of advice to users.)

Rationale. In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (End of rationale.)
CHAPTER 6. GROUPS, CONTEXTS, COMMUNICATIONS, AND CACHING

One difficulty is the potential for size differences between Fortran integers and C pointers. To overcome this problem with attribute caching on communicators, functions are also given for this case. The functions to cache on datatypes and windows also address this issue. For a general discussion of the address size problem, see Section 16.3.6.

Advice to implementors. High-quality implementations should raise an error when a keyval that was created by a call to MPI.xxx_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI.xxx_GET_ATTR, MPI.xxx_SET_ATTR, MPI.xxx_DELETE_ATTR, or MPI.xxx_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.)

6.7.1 Functionality

Attributes can be attached to communicators, windows, and datatypes. Attributes are local to the process and specific to the communicator to which they are attached. Attributes are not propagated by MPI from one communicator to another except when the communicator is duplicated using MPI.COMM_DUP (and even then the application must give specific permission through callback functions for the attribute to be copied).

Advice to users. Attributes in C are of type void *. Typically, such an attribute will be a pointer to a structure that contains further information, or a handle to an MPI object. In Fortran, attributes are of type INTEGER. Such an attribute can be a handle to an MPI object, or just an integer-valued attribute. (End of advice to users.)

Advice to implementors. Attributes are scalar values, equal in size to, or larger than a C-language pointer. Attributes can always hold an MPI handle. (End of advice to implementors.)

The caching interface defined here requires that attributes be stored by MPI opaquely within a communicator, window, and datatype. Accessor functions include the following:

• obtain a key value (used to identify an attribute); the user specifies "callback" functions by which MPI informs the application when the communicator is destroyed or copied.

• store and retrieve the value of an attribute;

Advice to implementors. Caching and callback functions are only called synchronously, in response to explicit application requests. This avoids problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators. A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid

Advice to implementors. Caching and callback functions are only called synchronously, in response to explicit application requests. This avoids problems that result from repeated crossings between user and system space. (This synchronous calling rule is a general property of MPI.)

The choice of key values is under control of MPI. This allows MPI to optimize its implementation of attribute sets. It also avoids conflict between independent modules caching information on the same communicators. A much smaller interface, consisting of just a callback facility, would allow the entire caching facility to be implemented by portable code. However, with the minimal callback interface, some form of table searching is implied by the need to handle arbitrary communicators. In contrast, the more complete interface defined here permits rapid
access to attributes through the use of pointers in communicators (to find the attribute table) and cleverly chosen key values (to retrieve individual attributes). In light of the efficiency “hit” inherent in the minimal interface, the more complete interface defined here is seen to be superior. (End of advice to implementors.)

MPI provides the following services related to caching. They are all process local.

6.7.2 Communicators

Functions for caching on communicators are:

```c
typedef int MPI_Comm_create_keyval_function(MPI_Comm *comm, int comm_keyval, extra_state);

#define MPI_COMM_CREATE_KEYVAL(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval, extra_state)
IN comm_copy_attr_fn     copy callback function for comm_keyval (function)
IN comm_delete_attr_fn   delete callback function for comm_keyval (function)
OUT comm_keyval          key value for future access (integer)
IN extra_state           extra state for callback functions

int MPI_Comm_create_keyval(MPI_Comm *comm, int comm_keyval, extra_state)

#define MPI_COMM_DESTROY_KEYVAL(comm_keyval)
IN comm_keyval          key value for future access (integer)
IN extra_state           extra state for callback functions

int MPI_Comm_destroy_keyval(MPI_Comm *comm, int comm_keyval, extra_state)

static int MPI_Comm::create_keyval(MPI_Comm::Copy_attr_function* comm_copy_attr_fn,
                                      MPI_Comm::Delete_attr_function* comm_delete_attr_fn,
                                      void* extra_state)

Generates a new attribute key. Keys are locally unique in a process, and opaque to user, though they are explicitly stored in integers. Once allocated, the key value can be used to associate attributes and access them on any locally defined communicator.

This function replaces MPI_KEYVAL_CREATE, whose use is deprecated. The C binding is identical. The Fortran binding differs in that extra_state is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes.

The C callback functions are:

```c
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);
```

and

```c
typedef int MPI_Comm::delete_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                          void *attribute_val, void *extra_state);
```

The C callback functions are:

```c
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
                                          void *extra_state, void *attribute_val_in,
                                          void *attribute_val_out, int *flag);
```

and

```c
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                          void *extra_state);
```

Also, the copy and delete callback functions have Fortran bindings that are consistent with

This function replaces MPI_KEYVAL_CREATE, whose use is deprecated. The C binding is identical. The Fortran binding differs in that extra_state is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes.

The C callback functions are:

```c
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);
```

and

```c
typedef int MPI_Comm::delete_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                          void *attribute_val, void *extra_state);
```

The C callback functions are:

```c
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, int comm_keyval,
                                          void *extra_state, void *attribute_val_in,
                                          void *attribute_val_out, int *flag);
```

and

```c
typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                          void *extra_state);
```

Also, the copy and delete callback functions have Fortran bindings that are consistent with

This function replaces MPI_KEYVAL_CREATE, whose use is deprecated. The C binding is identical. The Fortran binding differs in that extra_state is an address-sized integer. Also, the copy and delete callback functions have Fortran bindings that are consistent with address-sized attributes.

The C callback functions are:

```c
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);
```

and

```c
typedef int MPI_Comm::delete_attr_function(MPI_Comm oldcomm, int comm_keyval,
                                          void *attribute_val, void *extra_state);
```
which are the same as the MPI-1.1 calls but with a new name. The old names are deprecated.

The Fortran callback functions are:

```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
  
  and

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

The C++ callbacks are:

```cpp
typedef int MPI::Comm::Copy_attr_function(const MPI::Comm& oldcomm,
   int comm_keyval, void* extra_state, void* attribute_val_in,
   void* attribute_val_out, bool& flag);

and

```cpp
typedef int MPI::Comm::Delete_attr_function(MPI::Comm& comm,
   int comm_keyval, void* attribute_val, void* extra_state);
```

The `COMM COPY` function is invoked when a communicator is duplicated by `MPI_COMM DUP`. The `COMM COPY` function should be of type `MPI_Comm copy function`. The copy callback function is invoked for each key value in oldcomm in arbitrary order. Each call to the copy callback is made with a key value and its corresponding attribute. If it returns `flag = 0`, then the attribute is deleted in the duplicated communicator. Otherwise (`flag = 1`), the new attribute value is set to the value returned in `attribute_val_out`. The function returns `MPI_SUCCESS` on success and an error code on failure (in which case `MPI_COMM DUP` will fail).

Advice to users. Even though both formal arguments `attribute_val_in` and `attribute_val_out` are of type `void*`, their usage differs. The C copy function is passed by MPI in `attribute_val_in` the value of the attribute, and in `attribute_val_out` the address of the attribute, so as to allow the function to return the (new) attribute value. The use of type `void*` for both is to avoid messy type casts.

A valid copy function is one that completely duplicates the information by making a full duplicate copy of the data structures implied by an attribute; another might just make another reference to that data structure, while using a reference-count mechanism. Other types of attributes might not copy at all (they might be specific to `oldcomm` only). (End of advice to users.)
Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (End of advice to implementors.)

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows.

The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_Comm_delete_attr.

comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C, C++, or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_COMM_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_KEYVAL_CREATE. Therefore, it can be used for static initialization of key values.

MPI_COMM_FREE_KEYVAL(comm_keyval)

INOUT comm_keyval key value (integer)

int MPI_Comm_free_keyval(int *comm_keyval)

MPI_COMM_FREE_KEYVAL(MPI_COMM_KEYVAL, IERROR)

INTEGER COMM_KEYVAL, IERROR

static void MPI::Comm::Free_keyval(int& comm_keyval)

Frees an extant attribute key. This function sets the value of keyval to MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explicitly freed by the program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance, or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed communicator.

This call is identical to the MPI-1 call MPI_KEYVAL_FREE but is needed to match the new communicator-specific creation function. The use of MPI_KEYVAL_FREE is deprecated.

Advice to implementors. A C interface should be assumed for copy and delete functions associated with key values created in C; a Fortran calling interface should be assumed for key values created in Fortran. (End of advice to implementors.)

Analogous to comm_copy_attr_fn is a callback deletion function, defined as follows.

The comm_delete_attr_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_Comm_delete_attr.

comm_delete_attr_fn should be of type MPI_Comm_delete_attr_function.

This function is called by MPI_COMM_FREE, MPI_COMM_DELETE_ATTR, and MPI_COMM_SET_ATTR to do whatever is needed to remove an attribute. The function returns MPI_SUCCESS on success and an error code on failure (in which case MPI_COMM_FREE will fail).

The argument comm_delete_attr_fn may be specified as MPI_COMM_NULL_DELETE_FN from either C, C++, or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_COMM_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_KEYVAL_CREATE. Therefore, it can be used for static initialization of key values.

MPI_COMM_FREE_KEYVAL(comm_keyval)

INOUT comm_keyval key value (integer)

int MPI_Comm_free_keyval(int *comm_keyval)

MPI_COMM_FREE_KEYVAL(MPI_COMM_KEYVAL, IERROR)

INTEGER COMM_KEYVAL, IERROR

static void MPI::Comm::Free_keyval(int& comm_keyval)

Frees an extant attribute key. This function sets the value of keyval to MPI_KEYVAL_INVALID. Note that it is not erroneous to free an attribute key that is in use, because the actual free does not transpire until after all references (in other communicators on the process) to the key have been freed. These references need to be explicitly freed by the program, either via calls to MPI_COMM_DELETE_ATTR that free one attribute instance, or by calls to MPI_COMM_FREE that free all attribute instances associated with the freed communicator.

This call is identical to the MPI-1 call MPI_KEYVAL_FREE but is needed to match the new communicator-specific creation function. The use of MPI_KEYVAL_FREE is deprecated.
MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

INOUT comm communicator from which attribute will be attached
  (handle)
IN comm_keyval key value (integer)
IN attribute_val attribute value

int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void *attribute_val)

MPI_COMM_SET_ATTR.COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR
INTEGER COMM, COMM_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
void MPI::Comm::Set_attr(int comm_keyval, const void* attribute_val) const

This function stores the stipulated attribute value attribute_val for subsequent retrieval
by MPI_COMM_GET_ATTR. If the value is already present, then the outcome is as if
MPI_COMM_DELETE_ATTR was first called to delete the previous value (and the callback
function comm_delete_attr_fn was executed), and a new value was next stored. The call
is erroneous if there is no key with value keyval; in particular MPI_KEYVAL_INVALID is an
erroneous key value. The call will fail if the comm_delete_attr_fn function returned an error
code other than MPI_SUCCESS.

This function replaces MPI_ATTR.Put, whose use is deprecated. The C binding is
identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_COMM_GET_ATTR(comm, comm_keyval, attribute_val, flag)

IN comm communicator to which the attribute is attached (handle)
IN comm_keyval key value (integer)
OUT attribute_val attribute value, unless flag = false
OUT flag false if no attribute is associated with the key (logical)

int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void *attribute_val,
  int *flag)

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
bool MPI::Comm::Get_attr(int comm_keyval, void* attribute_val) const

Retrieves attribute value by key. The call is erroneous if there is no key with value
keyval. On the other hand, the call is correct if the key value exists, but no attribute is
attached on comm for that key; in such case, the call returns flag = false. In particular
MPI_KEYVAL_INVALID is an erroneous key value.

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the value of
the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address of the
attribute value passed to MPI_Comm_set_attr. This is the equivalent of the C
attribute_ptr macro and the Fortran pointer attribute attribute value. In particular
MPI_KEYVAL_INVALID is an erroneous key value.

Advice to users. The call to MPI_Comm_set_attr passes in attribute_val the value of
the attribute; the call to MPI_Comm_get_attr passes in attribute_val the address of the
attribute value passed to MPI_Comm_set_attr. This is the equivalent of the C
attribute_ptr macro and the Fortran pointer attribute attribute value. In particular
MPI_KEYVAL_INVALID is an erroneous key value.
The new functions for caching on windows are:

6.7.3 Windows

The new functions for caching on windows are:

Rationale. The use of a formal parameter attribute_val or type void* (rather than void**) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void*. (End of rationale.)

This function replaces MPI_ATTR_GET, whose use is deprecated. The C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_COMM_DELETE_ATTR(comm, comm_keyval)
  INOUT comm communicator from which the attribute is deleted (handle)
  IN comm_keyval key value (integer)

int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)

Rationale. The use of a formal parameter attribute_val or type void* (rather than void**) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void*. (End of rationale.)

This function replaces MPI_ATTR_GET, whose use is deprecated. The C binding is identical. The Fortran binding differs in that attribute_val is an address-sized integer.

MPI_COMM_DELETE_ATTR(comm, comm_keyval)
  INOUT comm communicator from which the attribute is deleted (handle)
  IN comm_keyval key value (integer)

int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)

6.7. CACHING

The new functions for caching on windows are:

6.7.3 Windows

The new functions for caching on windows are:
MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

IN  win_copy_attr_fn  copy callback function for win_keyval (function)
IN  win_delete_attr_fn  delete callback function for win_keyval (function)
OUT  win_keyval  key value for future access (integer)
IN  extra_state  extra state for callback functions

The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN from either C, C++, or Fortran. MPI_WIN_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS.

MPI_WIN_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.

The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN from either C, C++, or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

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);

The Fortran callback functions are:

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
    and

SUBROUTINE WIN_DELETE_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
    and

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)

MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
    EXTRA_STATE, IERROR)
integer WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
integer INTEGER WIN_KEYVAL, IERROR
integer INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN from either C, C++, or Fortran. MPI_WIN_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS.

MPI_WIN_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.

The argument win_delete_attr_fn may be specified as MPI_WIN_NULL_DELETE_FN from either C, C++, or Fortran. MPI_WIN_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

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);

The Fortran callback functions are:

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
    and

SUBROUTINE WIN_DELETE_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
    and
The C++ callbacks are:

typedef int MPI::Win::Copy_attr_function(const MPI::Win& oldwin, int win_keyval, void* extra_state, void* attribute_val_in, void* attribute_val_out, bool& flag);

and
typedef int MPI::Win::Delete_attr_function(MPI::Win& win, int win_keyval, void* attribute_val, void* extra_state);

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_WIN_FREE), is erroneous.

MPI_WIN_FREE_KEYVAL(win_keyval)
INOUT win_keyval key value (integer)

int MPI_Win_free_keyval(int *win_keyval)

MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
INTEGER WIN_KEYVAL, IERROR
static void MPI::Win::Free_keyval(int& win_keyval)

MPI_WIN_SET_ATTR(win, win_keyval, attribute_val)
INOUT win window to which attribute will be attached (handle)
IN win_keyval key value (integer)
IN attribute_val attribute value

int MPI_Win_set_attr(MPI_Win win, int win_keyval, void *attribute_val)

MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
INTEGER WIN, WIN_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

void MPI::Win::Set_attr(int win_keyval, const void* attribute_val)
6.7.4 Datatypes
The new functions for caching on datatypes are:

```c
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)
```

```c
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL, EXTRA_STATE, IERROR)
```

```c
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)
```

```c
MPI_TYPE_CREATE_KEYVAL(TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN, TYPE_KEYVAL, EXTRA_STATE, IERROR)
```
6.7. CACHING

EXTERNAL TYPE_COPY_ATTR_FN, TYPE_DELETE_ATTR_FN
INTEGER TYPE_KEYVAL, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

static int MPI::Datatype::Create_keyval(MPI::Datatype::Copy_attr_function* type_copy_attr_fn, MPI::Datatype::Delete_attr_function* type_delete_attr_fn, void* extra_state)

The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or MPI_TYPE_DUP_FN from either C, C++, or Fortran. MPI_TYPE_NULL_COPY_FN is a function that does nothing other than returning flag = 0 and MPI_SUCCESS.

MPI_TYPE_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.

The argument type_delete_attr_fn may be specified as MPI_TYPE_NULL_DELETE_FN from either C, C++, or Fortran. MPI_TYPE_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS.

The C callback functions are:

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);

and

typedef int MPI_Type_delete_attr_function(MPI::Datatype oldtype, int type_keyval, void* extra_state, void *attribute_val_in, void *attribute_val_out, bool& flag);

The Fortran callback functions are:

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

and

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

The C++ callbacks are:

typedef int MPI::Datatype::Copy_attr_function(const MPI::Datatype& oldtype, int type_keyval, void* extra_state, const void* attribute_val_in, void* attribute_val_out, bool& flag);

and

typedef int MPI::Datatype::Delete_attr_function(MPI::Datatype& type, int type_keyval, void* attribute_val, void* extra_state);

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_TYPE_FREE),...
is erroneous.

```c
MPI_TYPE_FREE_KEYVAL(type_keyval)
INOUT type_keyval key value (integer)
int MPI_Type_free_keyval(int *type_keyval)
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
INTEGER TYPE_KEYVAL, IERROR
static void MPI::Datatype::Free_keyval(int& type_keyval)

MPI_TYPE_SET_ATTR(type, type_keyval, attribute_val)
INOUT type datatype to which attribute will be attached (handle)
IN type_keyval key value (integer)
IN attribute_val attribute value
int MPI_Type_set_attr(MPI_Datatype type, int type_keyval,
    void *attribute_val)
MPI_TYPE_SET_ATTR(TYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
INTEGER TYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
void MPI::Datatype::Set_attr(int type_keyval, const void* attribute_val)

MPI_TYPE_GET_ATTR(type, type_keyval, attribute_val, flag)
IN type datatype to which the attribute is attached (handle)
IN type_keyval key value (integer)
OUT attribute_val attribute value, unless flag = false
OUT flag false if no attribute is associated with the key (logical)
int MPI_Type_get_attr(MPI_Datatype type, int type_keyval,
    void *attribute_val, int *flag)
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
bool MPI::Datatype::Get_attr(int type_keyval, void* attribute_val) const
```
MPI_TYPE_DELETE_ATTR(type, type_keyval)
INOUT type datatype from which the attribute is deleted (handle)
IN type_keyval key value (integer)

int MPI_Type_delete_attr(MPI_Datatype type, int type_keyval)
MPI_TYPE_DELETE_ATTR(TYPE, TYPE_KEYVAL, IERROR)
INTEGER TYPE, TYPE_KEYVAL, IERROR
void MPI::Datatype::Delete_attr(int type_keyval)

6.7.6 Attributes Example

Advice to users. This example shows how to write a collective communication operation that uses caching to be more efficient after the first call. The coding style assumes that MPI function results return only error statuses. (*End of advice to users.*)

/* key for this module's stuff: */
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct gop_stuff_type {
  int ref_count;  /* reference count */
  /* other stuff, whatever else we want */
} gop_stuff_type;

Efficient_Collective_Op (comm, ...)
MPI_Comm comm;
{
  gop_stuff_type *gop_stuff;
  MPI_Group group;
  int foundflag;
  MPI_Comm_group(comm, &group);
}
if (gop_key == MPI_KEYVAL_INVALID) /* get a key on first call ever */
{
    if ( ! MPI_Comm_create_keyval( gop_stuff_copier,
        gop_stuff_destructor, &gop_key, (void *)(0));
        /* get the key while assigning its copy and delete callback
        behavior. */
    MPI_Abort (comm, 99);
}

MPI_Comm_get_attr (comm, gop_key, &gop_stuff, &foundflag);
if (foundflag){ /* This module has executed in this group before.
    We will use the cached information */
} else{ /* This is a group that we have not yet cached anything in.
    We will now do so.
    */
    /* First, allocate storage for the stuff we want,
    and initialize the reference count */
    gop_stuff = (gop_stuff_type *) malloc (sizeof(gop_stuff_type));
    if (gop_stuff == NULL) { /* abort on out-of-memory error */ }
    gop_stuff -> ref_count = 1;/* Second, fill in *gop_stuff with whatever we want.
      This part isn't shown here */
    /* Third, store gop_stuff as the attribute value */
    MPI_Comm_set_attr ( comm, gop_key, gop_stuff);
    /* Then, in any case, use contents of *gop_stuff
    to do the global op ... */
}

/* The following routine is called by MPI when a group is freed */
gop_stuff_destructor (comm, keyval, gop_stuff, extra)
MPI_Comm comm;
int keyval;
gop_stuff_type *gop_stuff;
void *extra;
{ if (keyval != gop_key) { /* abort -- programming error */ }

6.8. Naming Objects

There are many occasions on which it would be useful to allow a user to associate a printable identifier with an MPI communicator, window, or datatype, for instance error reporting, debugging, and profiling. The names attached to opaque objects do not propagate when the object is duplicated or copied by MPI routines. For communicators this can be achieved using the following two functions.

**MPI_COMM_SET_NAME** (comm, comm_name)

**INOUT** comm communicator whose identifier is to be set (handle)

**IN** comm_name the character string which is remembered as the name (string)

```c
int MPI_Comm_set_name(MPI_Comm comm, char *comm_name);
```

**MPI_COMM_SET_NAME** allows a user to associate a name string with a communicator. The character string which is passed to MPI_COMM_SET_NAME will be saved inside the communicator. The character string which is passed to MPI_COMM_SET_NAME will be saved inside the communicator.
MPI library (so it can be freed by the caller immediately after the call, or allocated on the stack). Leading spaces in name are significant but trailing ones are not.

MPI_Comm_set_name() is a local (non-collective) operation, which only affects the name of the communicator as seen in the process which made the MPI_Comm_set_name call. There is no requirement that the same (or any) name be assigned to a communicator in every process where it exists.

Advice to users. Since MPI_Comm_set_name() is provided to help debug code, it is sensible to give the same name to a communicator in all of the processes where it exists, to avoid confusion.  

(End of advice to users.)

The length of the name which can be stored is limited to the value of MPI_MAX_OBJECT_NAME in Fortran and MPI_MAX_OBJECT_NAME-1 in C and C++ to allow for the null terminator. Attempts to put names longer than this will result in truncation of the name. MPI_MAX_OBJECT_NAME must have a value of at least 64.

Advice to users. Under circumstances of store exhaustion an attempt to put a name of any length could fail, therefore the value of MPI_MAX_OBJECT_NAME should be viewed only as a strict upper bound on the name length, not a guarantee that setting names of less than this length will always succeed.  

(End of advice to users.)

Advice to implementors. Implementations which pre-allocate a fixed size space for a name should use the length of that allocation as the value of MPI_MAX_OBJECT_NAME. Implementations which allocate space for the name from the heap should still define MPI_MAX_OBJECT_NAME to be a relatively small value, since the user has to allocate space for a string of up to this size when calling MPI_Comm_get_name.  

(End of advice to implementors.)

MPI_COMM_GET_NAME (comm, comm_name, resultlen)

IN  comm    communicator whose name is to be returned (handle)
OUT  comm_name  the name previously stored on the communicator, or an empty string if no such name exists (string)
OUT  resultlen  length of returned name (integer)

int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)

MPI_COMM_GET_NAME(comm, comm_name, resultlen)

IN  comm    communicator whose name is to be returned (handle)
OUT  comm_name  the name previously stored on the communicator, or an empty string if no such name exists (string)
OUT  resultlen  length of returned name (integer)

int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)

MPI_COMM_GET_NAME returns the last name which has previously been associated with the given communicator. The name may be set and got from any language. The same name will be returned independent of the language used. name should be allocated so that it can hold a resulting string of length MPI_MAX_OBJECT_NAME characters.

MPI_COMM_GET_NAME returns a copy of the set name in name. In C, a null character is
6.8. NAMING OBJECTS

additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT.

If the user has not associated a name with a communicator, or an error occurs, MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, "" in C and C++). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

Rationale. We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.
- To make the attribute key useful additional code to call strdup is necessary. If this is not standardized then users have to write it. This is extra unneeded work which we can easily eliminate.
- The Fortran binding is not trivial to write (it will depend on details of the Fortran compilation system), and will not be portable. Therefore it should be in the library rather than in user code.

(End of rationale.)

Advice to users. The above definition means that it is safe simply to print the string returned by MPI_COMM_GET_NAME, as it is always a valid string even if there was no name.

Note that associating a name with a communicator has no effect on the semantics of an MPI program, and will (necessarily) increase the store requirement of the program, since the names must be saved. Therefore there is no requirement that users use these functions to associate names with communicators. However debugging and profiling MPI applications may be made easier if names are associated with communicators, since the debugger or profiler should then be able to present information in a less cryptic manner. (End of advice to users.)

The following functions are used for setting and getting names of datatypes.

MPI_TYPE_SET_NAME (type, type_name)

INOUT type

 datatype whose identifier is to be set (handle)

 IN type_name

 the character string which is remembered as the name (string)

 int MPI_Type_set_name(MPI_Datatype type, char *type_name)
MPI_TYPE_SET_NAME(TYPE, TYPE_NAME, IERROR)
INTEGER TYPE, IERROR
CHARACTER(*) TYPE_NAME
void MPI::Datatype::Set_name(const char* type_name)

MPI_TYPE_GET_NAME (type, type_name, resultlen)
IN  type  datatype whose name is to be returned (handle)
OUT type_name  the name previously stored on the datatype, or a empty
          string if no such name exists (string)
OUT resultlen  length of returned name (integer)

int MPI_Type_get_name(MPI_Datatype type, char *type_name, int *resultlen)
MPI_TYPE_GET_NAME(TYPE, TYPE_NAME, RESULTLEN, IERROR)
INTEGER TYPE, RESULTLEN, IERROR
CHARACTER(*) TYPE_NAME

void MPI::Datatype::Get_name(char* type_name, int& resultlen) const
Named predefined datatypes have the default names of the data type name. For example, MPI_WCHAR has the default name of MPI_WCHAR.
The following functions are used for setting and getting names of windows.

MPI_WIN_SET_NAME (win, win_name)
INOUT win  window whose identifier is to be set (handle)
IN  win_name  the character string which is remembered as the name
              (string)

int MPI_Win_set_name(MPI_Win win, char *win_name)
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
INTEGER WIN, IERROR
CHARACTER(*) WIN_NAME

void MPI::Win::Set_name(const char* win_name)

MPI_WIN_GET_NAME (win, win_name, resultlen)
IN  win  window whose name is to be returned (handle)
OUT win_name  the name previously stored on the window, or a empty
             string if no such name exists (string)
OUT resultlen  length of returned name (integer)

int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
6.9. FORMALIZING THE LOOSELY SYNCHRONOUS MODEL

In this section, we make further statements about the loosely synchronous model, with particular attention to intra-communication.

6.9.1 Basic Statements

When a caller passes a communicator (that contains a context and group) to a callee, that communicator must be free of side effects throughout execution of the subprogram: there should be no active operations on that communicator that might involve the process. This provides one model in which libraries can be written, and work “safely.” For libraries so designated, the callee has permission to do whatever communication it likes with the communicator, and under the above guarantee knows that no other communications will interfere. Since we permit good implementations to create new communicators without synchronization (such as by preallocated contexts on communicators), this does not impose a significant overhead.

This form of safety is analogous to other common computer-science usages, such as passing a descriptor of an array to a library routine. The library routine has every right to expect such a descriptor to be valid and modifiable.

6.9.2 Models of Execution

In the loosely synchronous model, transfer of control to a parallel procedure is effected by having each executing process invoke the procedure. The invocation is a collective operation: it is executed by all processes in the execution group, and invocations are similarly ordered at all processes. However, the invocation need not be synchronized.

We say that a parallel procedure is active in a process if the process belongs to a group that may collectively execute the procedure, and some member of that group is currently executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure.

Static communicator allocation

This covers the case where, at any point in time, at most one invocation of a parallel procedure can be active at any process, and the group of executing processes is fixed. For example, all invocations of parallel procedures involve all processes, processes are single-threaded, and there are no recursive invocations.

In such a case, a communicator can be statically allocated to each procedure. The static allocation can be done in a preamble, as part of initialization code. If the parallel procedures can be organized into libraries, so that only one procedure of each library can
be concurrently active in each processor, then it is sufficient to allocate one communicator per library.

Dynamic communicator allocation

Calls of parallel procedures are well-nested if a new parallel procedure is always invoked in a subset of a group executing the same parallel procedure. Thus, processes that execute the same parallel procedure have the same execution stack.

In such a case, a new communicator needs to be dynamically allocated for each new invocation of a parallel procedure. The allocation is done by the caller. A new communicator can be generated by a call to MPI_COMM_DUP, if the callee execution group is identical to the caller execution group, or by a call to MPI_COMM_SPLIT if the caller execution group is split into several subgroups executing distinct parallel routines. The new communicator is passed as an argument to the invoked routine.

The need for generating a new communicator at each invocation can be alleviated or avoided altogether in some cases: If the execution group is not split, then one can allocate a stack of communicators in a preamble, and next manage the stack in a way that mimics the stack of recursive calls.

One can also take advantage of the well-ordering property of communication to avoid confusing caller and callee communication, even if both use the same communicator. To do so, one needs to abide by the following two rules:

• messages sent before a procedure call (or before a return from the procedure) are also received before the matching call (or return) at the receiving end;
• messages are always selected by source (no use is made of MPI_ANY_SOURCE).

The General case

In the general case, there may be multiple concurrently active invocations of the same parallel procedure within the same group; invocations may not be well-nested. A new communicator needs to be created for each invocation. It is the user’s responsibility to make sure that, should two distinct parallel procedures be invoked concurrently on overlapping sets of processes, then communicator creation be properly coordinated.
Chapter 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 6, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the “virtual topology.”

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal only with machine-independent mapping.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [32]. On the other hand, if there is no way for the user to specify the logical process arrangement as a “virtual topology,” a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [10, 11].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and

Chapter 7

Process Topologies

7.1 Introduction

This chapter discusses the MPI topology mechanism. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware.

As stated in Chapter 6, a process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the underlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement is described by a graph. In this chapter we will refer to this logical process arrangement as the “virtual topology.”

A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal only with machine-independent mapping.

Rationale. Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [32]. On the other hand, if there is no way for the user to specify the logical process arrangement as a “virtual topology,” a random mapping is most likely to result. On some machines, this will lead to unnecessary contention in the interconnection network. Some details about predicted and measured performance improvements that result from good process-to-processor mapping on modern wormhole-routing architectures can be found in [10, 11].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and
CHAPTER 7. PROCESS TOPOLOGIES

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a “missing link” in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping. Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

Rationale. Experience with similar techniques in PARMAACS [5, 9] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (End of rationale.)

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem than that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a $(2 \times 2)$ grid is as follows.

- coord $(0,0)$: rank 0
- coord $(0,1)$: rank 1
- coord $(1,0)$: rank 2
- coord $(1,1)$: rank 3

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.

notational power in message-passing programming. (End of rationale.)

7.2 Virtual Topologies

The communication pattern of a set of processes can be represented by a graph. The nodes represent processes, and the edges connect processes that communicate with each other. MPI provides message-passing between any pair of processes in a group. There is no requirement for opening a channel explicitly. Therefore, a “missing link” in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that this connection is neglected in the virtual topology. This strategy implies that the topology gives no convenient way of naming this pathway of communication. Another possible consequence is that an automatic mapping tool (if one exists for the runtime environment) will not take account of this edge when mapping. Edges in the communication graph are not weighted, so that processes are either simply connected or not connected at all.

Rationale. Experience with similar techniques in PARMAACS [5, 9] show that this information is usually sufficient for a good mapping. Additionally, a more precise specification is more difficult for the user to set up, and it would make the interface functions substantially more complicated. (End of rationale.)

Specifying the virtual topology in terms of a graph is sufficient for all applications. However, in many applications the graph structure is regular, and the detailed set-up of the graph would be inconvenient for the user and might be less efficient at run time. A large fraction of all parallel applications use process topologies like rings, two- or higher-dimensional grids, or tori. These structures are completely defined by the number of dimensions and the numbers of processes in each coordinate direction. Also, the mapping of grids and tori is generally an easier problem than that of general graphs. Thus, it is desirable to address these cases explicitly.

Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a $(2 \times 2)$ grid is as follows.

- coord $(0,0)$: rank 0
- coord $(0,1)$: rank 1
- coord $(1,0)$: rank 2
- coord $(1,1)$: rank 3

7.3 Embedding in MPI

The support for virtual topologies as defined in this chapter is consistent with other parts of MPI, and, whenever possible, makes use of functions that are defined elsewhere. Topology information is associated with communicators. It is added to communicators using the caching mechanism described in Chapter 6.
7.4 Overview of the Functions

The functions MPI_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph) virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. All input arguments must have identical values on all processes of the group of comm_old. A new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

**Rationale.** Similar functions are contained in EXPRESS [12] and PARMACS. (End of rationale.)

The function MPI_TOPO_TEST can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET, for general graphs, and MPI_CARTDIM_GET and MPI_CART_GET, for Cartesian topologies. Several additional functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). The function MPI_CART_SHIFT provides the information needed to communicate with neighbors in a Cartesian dimension. The two functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to extract the neighbors of a node in a graph. The function MPI_CART_SUB is collective over the input communicator’s group; all other functions are local.

Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.7 outlines such an implementation.

7.4 Overview of the Functions

The functions MPI_GRAPH_CREATE and MPI_CART_CREATE are used to create general (graph) virtual topologies and Cartesian topologies, respectively. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. All input arguments must have identical values on all processes of the group of comm_old. A new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 6). In analogy to function MPI_COMM_CREATE, no cached information propagates from comm_old to comm_topol.

MPI_CART_CREATE can be used to describe Cartesian structures of arbitrary dimension. For each coordinate direction one specifies whether the process structure is periodic or not. Note that an n-dimensional hypercube is an n-dimensional torus with 2 processes per coordinate direction. Thus, special support for hypercube structures is not necessary. The local auxiliary function MPI_DIMS_CREATE can be used to compute a balanced distribution of processes among a given number of dimensions.

**Rationale.** Similar functions are contained in EXPRESS [12] and PARMACS. (End of rationale.)

The function MPI_TOPO_TEST can be used to inquire about the topology associated with a communicator. The topological information can be extracted from the communicator using the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET, for general graphs, and MPI_CARTDIM_GET and MPI_CART_GET, for Cartesian topologies. Several additional functions are provided to manipulate Cartesian topologies: the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice versa; the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). The function MPI_CART_SHIFT provides the information needed to communicate with neighbors in a Cartesian dimension. The two functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to extract the neighbors of a node in a graph. The function MPI_CART_SUB is collective over the input communicator’s group; all other functions are local.

Two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP are presented in the last section. In general these functions are not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.7 outlines such an implementation.
7.5 Topology Constructors

7.5.1 Cartesian Constructor

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN comm_old input communicator (handle)
IN ndims integer of dimensions of Cartesian grid (integer)
IN dims integer array of size ndims specifying the number of processes in each dimension
IN periods logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
IN reorder ranking may be reordered (false) or not (true) (logical)
OUT comm_cart communicator with new Cartesian topology (handle)

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)

INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER

MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[], const bool periods[], bool reorder) const

MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If reorder = false then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of comm, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created.

The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.

7.5.2 Cartesian Convenience Function: MPI_DIMS_CREATE

For Cartesian topologies, the function MPI_DIMS_CREATE helps the user select a balanced distribution of processes per coordinate direction, depending on the number of processes in the group to be balanced and optional constraints that can be specified by the user. One use is to partition all the processes (the size of MPI_COMM_WORLD’s group) into an n-dimensional topology.
7.5. TOPOLOGY CONSTRUCTORS

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN nnodes number of nodes in a grid (integer)
IN ndims number of Cartesian dimensions (integer)
INOUT dims integer array of size ndims specifying the number of
nodes in each dimension

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions
and a total of nnodes nodes. The dimensions are set to be as close to each other as possible,
using an appropriate divisibility algorithm. The caller may further constrain the operation
of this routine by specifying elements of array dims. If dims[i] is set to a positive number,
the routine will not modify the number of nodes in dimension i; only those entries where
dims[i] = 0 are modified by the call.

Negative input values of dims[i] are erroneous. An error will occur if nnodes is not a
multiple of \( \prod_{i \in \text{dims}} \text{dims}[i] \).

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array
dims is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is
local.

Example 7.1

<table>
<thead>
<tr>
<th>dims before call</th>
<th>function call</th>
<th>dims on return</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>MPI_DIMS_CREATE(6, 2, dims)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>MPI_DIMS_CREATE(7, 2, dims)</td>
<td>(7,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_DIMS_CREATE(6, 3, dims)</td>
<td>(2,3,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_DIMS_CREATE(7, 3, dims)</td>
<td>erroneous call</td>
</tr>
</tbody>
</table>

int MPI_Dims_create(int nnodes, int ndims, int *dims)

INTEGER NNODES, NDIMS, DIMS(*), IERROR

void MPI::Compute_dims(int nnodes, int ndims, int dims[])

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions
and a total of nnodes nodes. The dimensions are set to be as close to each other as possible,
using an appropriate divisibility algorithm. The caller may further constrain the operation
of this routine by specifying elements of array dims. If dims[i] is set to a positive number,
the routine will not modify the number of nodes in dimension i; only those entries where
dims[i] = 0 are modified by the call.

Negative input values of dims[i] are erroneous. An error will occur if nnodes is not a
multiple of \( \prod_{i \in \text{dims}} \text{dims}[i] \).

For dims[i] set by the call, dims[i] will be ordered in non-increasing order. Array
dims is suitable for use as input to routine MPI_CART_CREATE. MPI_DIMS_CREATE is
local.

Example 7.1

<table>
<thead>
<tr>
<th>dims before call</th>
<th>function call</th>
<th>dims on return</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>MPI_DIMS_CREATE(6, 2, dims)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>MPI_DIMS_CREATE(7, 2, dims)</td>
<td>(7,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_DIMS_CREATE(6, 3, dims)</td>
<td>(2,3,1)</td>
</tr>
<tr>
<td>(0,3,0)</td>
<td>MPI_DIMS_CREATE(7, 3, dims)</td>
<td>erroneous call</td>
</tr>
</tbody>
</table>
7.5.3 General (Graph) Constructor

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

IN    comm_old           input communicator (handle)
IN    nnodes             number of nodes in graph (integer)
IN    index              array of integers describing node degrees (see below)
IN    edges              array of integers describing graph edges (see below)
IN    reorder            ranking may be reordered (true) or not (false) (logical)
OUT   comm_graph         communicator with graph topology added (handle)

Example 7.2 Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
Then, the input arguments are:

- `nnodes = 4`
- `index = [2, 3, 4, 6]`
- `edges = [1, 3, 0, 2]`

Thus, in C, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i`, `i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges[0]`, for `0 ≤ j ≤ index[0] - 1` and the list of neighbors of node `i`, `i > 0`, is stored in `edges[0]`, `index[i-1] ≤ j ≤ index[i] - 1`.

In Fortran, `index(1)` is the degree of node zero, and `index(i+1) - index(i)` is the degree of node `i`, `i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges(0)`, for `1 ≤ j ≤ index(1)` and the list of neighbors of node `i`, `i > 0`, is stored in `edges(j)`, `index(i) + 1 ≤ j ≤ index(i+1)`.

A single process is allowed to be defined multiple times in the list of neighbors of a process (i.e., there may be multiple edges between two processes). A process is also allowed to be a neighbor to itself (i.e., a self loop in the graph). The adjacency matrix is allowed to be 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.)*

**Advice to implementors.** The following topology information is likely to be stored with a communicator:

- Type of topology (Cartesian/graph),
- For a Cartesian topology:
  1. `ndims` (number of dimensions),
  2. `dims` (numbers of processes per coordinate direction),
  3. `periods` (periodicity information),
  4. `own_position` (own position in grid, could also be computed from rank and dims).
- For a graph topology:
  1. `index`,
  2. `edges`,
  which are the vectors defining the graph structure.

For a graph structure the number of nodes is equal to the number of processes in the group. Therefore, the number of nodes does not have to be stored explicitly. An additional zero entry at the start of array `index` simplifies access to the topology information. *(End of advice to implementors.)*
7.5.4 Topology Inquiry Functions

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

MPI_TOPO_TEST(comm, status)
IN comm communicator (handle)
OUT status topology type of communicator comm (state)

int MPI_Topo_test(MPI_Comm comm, int *status)
MPI_TOPO_TEST(COMM, STATUS, IERROR)

The function MPI_TOPO_TEST returns the type of topology that is assigned to a communicator.

The output value status is one of the following:

- MPI_GRAPH graph topology
- MPI_CART Cartesian topology
- MPI_UNDEFINED no topology

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)
IN comm communicator for group with graph structure (handle)
OUT nnodes number of nodes in graph (integer) (same as number of processes in the group)
OUT nedges number of edges in graph (integer)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)

void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const

Functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET retrieve the graph-topology information that was associated with a communicator by MPI_GRAPH_CREATE.

The information provided by MPI_GRAPHDIMS_GET can be used to dimension the vectors index and edges correctly for the following call to MPI_GRAPH_GET.
7.5. TOPOLOGY CONSTRUCTORS

**MPI_GRAPH_GET**

```c
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges)
```

- **IN** `comm`: communicator with graph structure (handle)
- **IN** `maxindex`: length of vector index in the calling program (integer)
- **IN** `maxedges`: length of vector edges in the calling program (integer)
- **OUT** `index`: array of integers containing the graph structure (for details see the definition of MPI_GRAPH_CREATE)
- **OUT** `edges`: array of integers containing the graph structure

**MPI_CARTDIM_GET**

```c
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

- **IN** `comm`: communicator with Cartesian structure (handle)
- **OUT** `ndims`: number of dimensions of the Cartesian structure (integer)

The functions MPI_CARTDIM_GET and MPI_CART_GET return the Cartesian topology information that was associated with a communicator by MPI_CART_CREATE. 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.
MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN comm communicator with Cartesian structure (handle)
IN maxdims length of vectors dims, periods, and coords in the calling program (integer)
OUT dims number of processes for each Cartesian dimension (array of integer)
OUT periods periodicity (true/false) for each Cartesian dimension (array of logical)
OUT coords coordinates of calling process in Cartesian structure (array of integer)

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
LOGICAL PERIODS(*)

void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[], int coords[])
const

MPI_CART_RANK(comm, coords, rank)

IN comm communicator with Cartesian structure (handle)
IN coords integer array (of size ndims) specifying the Cartesian coordinates of a process
OUT rank rank of specified process (integer)

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
INTEGER COMM, COORDS(*), RANK, IERROR

int MPI::Cartcomm::Get_cart_rank(const int coords[])
const

For a process group with Cartesian structure, the function MPI_CART_RANK translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension i with periods(i) = true, if the coordinate, coords(i), is out of range, that is, coords(i) < 0 or coords(i) ≥ dims(i), it is shifted back to the interval 0 ≤ coords(i) < dims(i) automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

If comm is associated with a zero-dimensional Cartesian topology, coord is not significant and 0 is returned in rank.
7.5. TOPOLOGY CONSTRUCTORS

**MPI_CART_COORDS**

| IN  | comm          | communicator with Cartesian structure (handle) |
| IN  | rank          | rank of a process within group of comm (integer) |
| IN  | maxdims       | length of vector coords in the calling program (integer) |
| OUT | coords        | integer array (of size ndims) containing the Cartesian coordinates of specified process (array of integers) |

```c
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

The inverse mapping, rank-to-coordinates translation is provided by **MPI_CART_COORDS**.

If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

**MPI_GRAPH_NEIGHBORS_COUNT**

| IN  | comm          | communicator with graph topology (handle) |
| IN  | rank          | rank of process in group of comm (integer) |
| OUT | nneighbors    | number of neighbors of specified process (integer) |

```c
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

The inverse mapping, rank-to-coordinates translation is provided by **MPI_CART_COORDS**.

If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

**MPI_GRAPH_NEIGHBORS**

| IN  | comm          | communicator with graph topology (handle) |
| IN  | rank          | rank of process in group of comm (integer) |
| IN  | maxneighbors  | size of array neighbors (integer) |
| OUT | neighbors     | ranks of processes that are neighbors to specified process (array of integers) |

```c
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)
```

**MPI_GRAPH_NEIGHBORS_COUNT** and **MPI_GRAPH_NEIGHBORS** provide adjacency information for a general graph topology.
Example 7.3 Suppose that \( \text{comm} \) is a communicator with a shuffle-exchange topology. The group has \( 2^n \) members. Each process is labeled by \( a_1, \ldots, a_n \), with \( a_i \in \{0, 1\} \), and has three neighbors: exchange\((a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, a_0 \) \((a = 1 - a)\), shuffle\((a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1\) and unshuffle\((a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}\). The graph adjacency list is illustrated below for \( n = 3\).

<table>
<thead>
<tr>
<th>node</th>
<th>exchange</th>
<th>shuffle</th>
<th>unshuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (000)</td>
<td>1 0 0</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 (001)</td>
<td>0 2 4</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>2 (010)</td>
<td>3 4 1</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>3 (011)</td>
<td>2 6 5</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>4 (100)</td>
<td>5 1 2</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>5 (101)</td>
<td>4 3 6</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>6 (110)</td>
<td>7 5 3</td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>7 (111)</td>
<td>6 7 7</td>
<td>0 0 1</td>
<td></td>
</tr>
</tbody>
</table>

Suppose that the communicator \( \text{comm} \) has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

C assume: each process has stored a real number \( A \).
C extract neighborhood information
C perform exchange permutation
C perform shuffle permutation
C perform unshuffle permutation

7.5.5 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI\_SENDRECV operation is likely to be used along a coordinate direction to perform a shift of data. As input, MPI\_SENDRECV takes the rank of a source process for the receive, and the rank of a destination process for the send. If the function MPI\_CART\_SHIFT is called for a Cartesian process group, it provides the calling process with the above identifiers, which then can be passed to MPI\_SENDRECV.
The user specifies the coordinate direction and the size of the step (positive or negative). The function is local.

**MPI_CART_SHIFT**

```c
IN comm communicator with Cartesian structure (handle)
IN direction coordinate dimension of shift (integer)
IN disp displacement (> 0: upwards shift, < 0: downwards shift) (integer)
OUT rank_source rank of source process (integer)
OUT rank_dest rank of destination process (integer)
```

```c
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
int &rank_source, int &rank_dest)
```

**Example 7.4**

The communicator, `comm`, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of `REALs` is stored one element per process, in variable `A`. One wishes to skew this array, by shifting column `i` (vertically, i.e., along the column) by `i` steps.

```c
.... C find process rank
CALL MPI_COMM_RANK(comm, rank, ierr)
C find Cartesian coordinates
CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
C compute shift source and destination
CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
C skew array
    CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm,
    + status, ierr)
```
In Fortran, the dimension indicated by \( \text{DIRECTION} = i \) has \( \text{DIMS}(i+1) \) nodes, where \( \text{DIMS} \) is the array that was used to create the grid. In C, the dimension indicated by \( \text{direction} = i \) is the dimension specified by \( \text{dims}[i] \). (End of advice to users.)

### 7.5.6 Partitioning of Cartesian structures

**MPI_CART_SUB**

\[
\begin{align*}
\text{IN} & \quad \text{comm} & \text{communicator with Cartesian structure (handle)} \\
\text{IN} & \quad \text{remain_dims} & \text{the } i\text{-th entry of } \text{remain_dims} \text{ specifies whether the} \\
& & \text{1\text{-th dimension is kept in the subgrid (true) or is drop-} \\
& & \text{ped (false) (logical vector)} \\
\text{OUT} & \quad \text{newcomm} & \text{communicator containing the subgrid that includes} \\
& & \text{the calling process (handle)} \\
\end{align*}
\]

\[
\text{int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)}
\]

If a Cartesian topology has been created with **MPI_CART_CREATE**, the function **MPI_CART_SUB** can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in \( \text{remain_dims} \) are false or \( \text{comm} \) is already associated with a zero-dimensional Cartesian topology then \( \text{newcomm} \) is associated with a zero-dimensional Cartesian topology. (This function is closely related to **MPI_COMM_SPLIT**.)

**Example 7.5** Assume that **MPI_CART_CREATE(..., comm)** has defined a \((2 \times 3 \times 4)\) grid. Let \( \text{remain_dims} = (\text{true, false, true}) \). Then a call to,

\[
\text{MPI_CART_SUB(comm, remain_dims, comm_new)},
\]

will create three communicators each with eight processes in a \(2 \times 4\) Cartesian topology. If \( \text{remain_dims} = (\text{false, false, true}) \) then the call to **MPI_CART_SUB(comm, remain_dims, comm_new)** will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.

### 7.5.7 Low-Level Topology Functions

The two additional functions introduced in this section can be used to implement all other topology functions. In general they will not be called by the user directly, unless he or she is creating additional virtual topology capability other than that provided by MPI.

**MPI_CART_SUB**

\[
\begin{align*}
\text{IN} & \quad \text{comm} & \text{communicator with Cartesian structure (handle)} \\
\text{IN} & \quad \text{remain_dims} & \text{the } i\text{-th entry of } \text{remain_dims} \text{ specifies whether the} \\
& & \text{1\text{-th dimension is kept in the subgrid (true) or is drop-} \\
& & \text{ped (false) (logical vector)} \\
\text{OUT} & \quad \text{newcomm} & \text{communicator containing the subgrid that includes} \\
& & \text{the calling process (handle)} \\
\end{align*}
\]

\[
\text{int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)}
\]

If a Cartesian topology has been created with **MPI_CART_CREATE**, the function **MPI_CART_SUB** can be used to partition the communicator group into subgroups that form lower-dimensional Cartesian subgrids, and to build for each subgroup a communicator with the associated subgrid Cartesian topology. If all entries in \( \text{remain_dims} \) are false or \( \text{comm} \) is already associated with a zero-dimensional Cartesian topology then \( \text{newcomm} \) is associated with a zero-dimensional Cartesian topology. (This function is closely related to **MPI_COMM_SPLIT**.)

**Example 7.5** Assume that **MPI_CART_CREATE(..., comm)** has defined a \((2 \times 3 \times 4)\) grid. Let \( \text{remain_dims} = (\text{true, false, true}) \). Then a call to,

\[
\text{MPI_CART_SUB(comm, remain_dims, comm_new)},
\]

will create three communicators each with eight processes in a \(2 \times 4\) Cartesian topology. If \( \text{remain_dims} = (\text{false, false, true}) \) then the call to **MPI_CART_SUB(comm, remain_dims, comm_new)** will create six non-overlapping communicators, each with four processes, in a one-dimensional Cartesian topology.
The corresponding new function for general graph structures is as follows.

```c
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods, int *newrank)
```

`MPI_CART_MAP` (comm, ndims, dims, periods, newrank)

- **IN** `comm` input communicator (handle)
- **IN** `ndims` number of dimensions of Cartesian structure (integer)
- **IN** `dims` integer array of size `ndims` specifying the number of processes in each coordinate direction
- **IN** `periods` logical array of size `ndims` specifying the periodicity specification in each coordinate direction
- **OUT** `newrank` reordered rank of the calling process; `MPI_UNDEFINED` if calling process does not belong to grid (integer)

```c
MPI_CART_MAP(COMM, NDIMS, DIMS(*), NEWRANK, IERROR)
```

Advice to implementors. The function `MPI_CART_CREATE` (comm, ndims, dims, periods, reorder, comm_cart), with `reorder = true` can be implemented by calling `MPI_CART_MAP` (comm, ndims, dims, periods, newrank), then calling `MPI_COMM_SPLIT` (comm, color, key, comm_cart), with color = 0 if `newrank` ≠ `MPI_UNDEFINED`, color = `MPI_UNDEFINED` otherwise, and key = `newrank`.

The function `MPI_CART_SUB` (comm, remain_dims, comm_new) can be implemented by a call to `MPI_COMM_SPLIT` (comm, color, key, comm_cart), using a single number encoding of the lost dimensions as color and a single number encoding of the preserved dimensions as key.

All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (End of advice to implementors.)

The corresponding new function for general graph structures is as follows.

```c
int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[], const bool periods[])
```

const `MPI_CART_MAP` (comm, ndims, dims, periods, newrank)

- **IN** `comm` input communicator (handle)
- **IN** `ndims` number of dimensions of Cartesian structure (integer)
- **IN** `dims` integer array of size `ndims` specifying the number of processes in each coordinate direction
- **IN** `periods` logical array of size `ndims` specifying the periodicity specification in each coordinate direction
- **OUT** `newrank` reordered rank of the calling process; `MPI_UNDEFINED` if calling process does not belong to grid (integer)

```c
MPI_CART_MAP(COMM, NDIMS, DIMS(*), NEWRANK, IERROR)
```

Advice to implementors. The function `MPI_CART_CREATE` (comm, ndims, dims, periods, reorder, comm_cart), with `reorder = true` can be implemented by calling `MPI_CART_MAP` (comm, ndims, dims, periods, newrank), then calling `MPI_COMM_SPLIT` (comm, color, key, comm_cart), with color = 0 if `newrank` ≠ `MPI_UNDEFINED`, color = `MPI_UNDEFINED` otherwise, and key = `newrank`.

The function `MPI_CART_SUB` (comm, remain_dims, comm_new) can be implemented by a call to `MPI_COMM_SPLIT` (comm, color, key, comm_new), using a single number encoding of the lost dimensions as color and a single number encoding of the preserved dimensions as key.

All other Cartesian topology functions can be implemented locally, using the topology information that is cached with the communicator. (End of advice to implementors.)

The corresponding new function for general graph structures is as follows.
7.6 An Application Example

Example 7.6 The example in Figure 7.1 shows how the grid definition and inquiry functions can be used in an application program. A partial differential equation, for instance the Poisson equation, is to be solved on a rectangular domain. First, the processes organize themselves in a two-dimensional structure. Each process then inquires about the ranks of its neighbors in the four directions (up, down, right, left). The numerical problem is solved by an iterative method, the details of which are hidden in the subroutine relax.

In each relaxation step each process computes new values for the solution grid function at all points owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the exchange subroutine might contain a call like MPI_SEND(...,neigh_rank(1),...) to send updated values to the left-hand neighbor (i-1,j).

The function MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

<table>
<thead>
<tr>
<th>IN</th>
<th>comm</th>
<th>input communicator (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>nnodes</td>
<td>number of graph nodes (integer)</td>
</tr>
<tr>
<td>IN</td>
<td>index</td>
<td>integer array specifying the graph structure, see MPI_GRAPH_CREATE</td>
</tr>
<tr>
<td>IN</td>
<td>edges</td>
<td>integer array specifying the graph structure</td>
</tr>
<tr>
<td>OUT</td>
<td>newrank</td>
<td>reordered rank of the calling process; MPI_UNDEFINED if the calling process does not belong to graph (integer)</td>
</tr>
</tbody>
</table>

Advice to implementors. The function MPI_GRAPH_CREATE(comm, nnodes, index, edges, reorder, comm_graph), with reorder = true can be implemented by calling MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank), then calling MPI_COMM_SPLIT(comm, color, key, comm_graph), with color = 0 if newrank ≠ MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank.

All other graph topology functions can be implemented locally, using the topology information that is cached with the communicator. (End of advice to implementors.)
integer ndims, num_neigh
logical reorder
parameter (ndims=2, num_neigh=4, reorder=.true.)
integer comm, comm_cart, dims(ndims), neigh_def(ndims), ierr
integer neigh_rank(num_neigh), own_position(ndims), i, j
logical periods(ndims)
real*8 u(0:101,0:101), f(0:101,0:101)
data dims / ndims 0 /
comm = MPI_COMM_WORLD
C Set process grid size and periodicity
call MPI_DIMS_CREATE(comm, ndims, dims,ierr)
periods(1) = .TRUE.
periods(2) = .TRUE.
C Create a grid structure in WORLD group and inquire about own position
call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart,ierr)
call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position,ierr)
C Look up the ranks for the neighbors. Own process coordinates are (i,j).
C Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1)
i = own_position(1)
j = own_position(2)
neigh_def(1) = i-1
neigh_def(2) = j
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1),ierr)
neigh_def(1) = i+1
neigh_def(2) = j
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2),ierr)
neigh_def(1) = i
neigh_def(2) = j-1
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3),ierr)
neigh_def(1) = j+1
neigh_def(2) = j
call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4),ierr)
C Initialize the grid functions and start the iteration
call init (u, f)
do 10 it=1,100
    call relax (u, f)
10 continue
C Exchange data with neighbor processes
call exchange (u, comm_cart, neigh_rank, num_neigh)
call output (u)
end

Figure 7.1: Set-up of process structure for two-dimensional parallel Poisson solver.
Chapter 8

MPI Environmental Management

This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

8.1 Implementation Information

8.1.1 Version Inquiries

In order to cope with changes to the MPI Standard, there are both compile-time and runtime ways to determine which version of the standard is in use in the environment one is using.

The “version” will be represented by two separate integers, for the version and sub-version: In C and C++,

```
#define MPI_VERSION 2
#define MPI_SUBVERSION 1
```

in Fortran,

```
INTEGER MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION = 2)
PARAMETER (MPI_SUBVERSION = 1)
```

For runtime determination,

```
MPI_GET_VERSION( version, subversion )

OUT version number (integer)
OUT subversion number (integer)
```

```
int MPI_Get_version(int *version, int *subversion)
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
void MPI::Get_version(int& version, int& subversion)
```
**MPI GET VERSION** is one of the few functions that can be called before MPI_INIT and after MPI_FINALIZE. Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are (2,1), (2,0), and (1,2).

### 8.1.2 Environmental Inquiries

A set of attributes that describe the execution environment are attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_ATTR_GET described in Chapter 6. It is erroneous to delete these attributes, free their keys, or change their values.

The list of predefined attribute keys include:

- **MPI_TAG_UB** Upper bound for tag value.
- **MPI_HOST** Host process rank, if such exists, MPI_PROC_NULL, otherwise.
- **MPI_PROC** rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.
- **MPI_LTIME_IS_GLOBAL** Boolean variable that indicates whether clocks are synchronized.

Vendors may add implementation specific parameters (such as node number, real memory size, virtual memory size, etc.)

These predefined attributes do not change value between MPI initialization (MPI_INIT and MPI completion (MPI_FINALIZE), and cannot be updated or deleted by users.

*Advice to users.* Note that in the C binding, the value returned by these attributes is a pointer to an int containing the requested value. (*End of advice to users.*)

The required parameter values are discussed in more detail below:

#### Tag Values

Tag values range from 0 to the value returned for MPI_TAG_UB inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be at least 32767. An MPI implementation is free to make the value of MPI_TAG_UB larger than this; for example, the value $2^{30} - 1$ is also a legal value for MPI_TAG_UB.

The attribute MPI_TAG_UB has the same value on all processes of MPI_COMM_WORLD.

#### Host Rank

The value returned for MPI_HOST gets the rank of the BST process in the group associated with communicator MPI_COMM_WORLD, if there is such. MPI_PROC_NULL is returned if there is no host. MPI does not specify what it means for a process to be a BST, nor does it requires that a BST exists.

The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.
8.1. IMPLEMENTATION INFORMATION

IO Rank

The value returned for MPI_Rank is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, READ, WRITE). For C and C++, this means that all of the ISO C and C++ I/O operations are supported (e.g., fopen, fprintf, lseek).

If every process can provide language-standard I/O, then the value MPI_RANK will be returned. Otherwise, if the calling process can provide language-standard I/O, then its rank will be returned. Otherwise, if some process can provide language-standard I/O then the rank of one such process will be returned. The same value need not be returned by all processes. If no process can provide language-standard I/O, then the value MPI_PROC_NULL will be returned.

Advice to users. Note that input is not collective, and this attribute does not indicate which process can or does provide input. (End of advice to users.)

Clock Synchronization

The value returned for MPI_WTIME is 1 if clocks at all processes in MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to MPI_WTIME, will be less than one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one.

The attribute MPI_WTIME need not be present when the clocks are not synchronized (however, the attribute key MPI_WTIME is always valid). This attribute may be associated with communicators other then MPI_COMM_WORLD.

The attribute MPI_WTIME has the same value on all processes of MPI_COMM_WORLD.

MPI_GET_PROCESSOR_NAME( name, resultlen )

OUT name A unique specifier for the actual (as opposed to virtual) node.

OUT resultlen Length (in printable characters) of the result returned in name

int MPI_Get_processor_name(char *name, int *resultlen)

MPI_GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)

CHARACTER*(*) NAME
INTEGER RESULTLEN, IERROR

void MPI::Get_processor_name(char* name, int* resultlen)

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include “processor 9 in rack 4 of mpp.cs.org” and “231” (where 231 is the actual processor number in the
running homogeneous system). The argument `name` must represent storage that is at least `MPI_MAX_PROCESSOR_NAME` characters long. `MPI_GET_PROCESSOR_NAME` may write up to this many characters into `name`.

The number of characters actually written is returned in the output argument, `resultlen`.

In C, a null character is additionally stored at `name[resultlen]`. The `resultlen` cannot be larger than `MPI_MAX_PROCESSOR_NAME`. In Fortran, name is padded on the right with blank characters. The `resultlen` cannot be larger than `MPI_MAX_PROCESSOR_NAME`.

### Rationale
This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI requires or defines process migration; this definition of `MPI_GET_PROCESSOR_NAME` simply allows such an implementation. (End of rationale.)

### Advice to users
The user must provide at least `MPI_MAX_PROCESSOR_NAME` space to write the processor name — processor names can be this long. The user should examine the output argument, `resultlen`, to determine the actual length of the name. (End of advice to users.)

The constant `MPI_BSEND_OVERHEAD` provides an upper bound on the fixed overhead per message buffered by a call to `MPI_BSEND` (see Section 3.6.1).

### 8.2 Memory Allocation
In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. However, implementations may restrict the use of the `MPI_WIN_LOCK` and `MPI_WIN_UNLOCK` functions to windows allocated in such memory (see Section 11.4.3.)

```c
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

MPI_ALLOC_MEM(size, info, baseptr)    
IN size    size of memory segment in bytes (nonnegative integer) 
IN info    info argument (handle) 
OUT baseptr pointer to beginning of memory segment allocated 
```

```c
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

MPI_ALLOC_MEM(size, info, baseptr, ierr) 
IN size    size of memory segment in bytes (nonnegative integer) 
IN info    info argument (handle) 
OUT baseptr pointer to beginning of memory segment allocated 
```

The `info` argument can be used to provide directives that control the desired location of the allocated memory. Such a directive does not affect the semantics of the call. Valid info values are implementation-dependent; a null directive value of `info = MPI_INFO_NULL` is always valid.

### MPI_ALLOC_MEM

```c
void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info) 
```

```c
void* MPI::Alloc_mem(MPI::Aint size, const MPI::Info& info) 
```

The `info` argument can be used to provide directives that control the desired location of the allocated memory. Such a directive does not affect the semantics of the call. Valid info values are implementation-dependent; a null directive value of `info = MPI_INFO_NULL` is always valid.
8.2. MEMORY ALLOCATION

The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM to indicate it failed because memory is exhausted.

```fortran
MPI_FREE_MEM(base)
IN  base  initial address of memory segment allocated by
MPI_ALLOC_MEM (choice)

int MPI_Free_mem(void *base)
MPI_FREE_MEM(BASE, IERROR)
[type] BASE(*) INTEGER IERROR
void MPI::Free_mem(void *base)

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

Rationale. The C and C++ bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar to the bindings for the malloc and free C library calls: a call to MPI_Alloc_mem(...) &base should be paired with a call to MPI_Free_mem(base) (one less level of indirection). Both arguments are declared to be of same type void* so as to facilitate type casting. The Fortran binding is consistent with the C and C++ bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (End of rationale.)

Advice to implementors. If MPI_ALLOC_MEM allocates special memory, then a design similar to the design of C malloc and free functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_ALLOC_MEM simply invokes malloc, and MPI_FREE_MEM invokes free.

A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (End of advice to implementors.)

Example 8.1 Example of use of MPI_ALLOC_MEM, in Fortran with pointer support. We assume 4-byte REALS, and assume that pointers are address-sized.

```fortran
REAL A
POINTER (P, A(100,100)) ! no memory is allocated
CALL MPI_ALLOC_MEM(4*100*100, MPI_INFO_NULL, P, IERR)
! memory is allocated

A(3,5) = 2.71;

CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```
```fortran
8.2. MEMORY ALLOCATION

The function MPI_ALLOC_MEM may return an error code of class MPI_ERR_NO_MEM to indicate it failed because memory is exhausted.

MPI_FREE_MEM(base)
IN  base  initial address of memory segment allocated by
MPI_ALLOC_MEM (choice)

int MPI_Free_mem(void *base)
MPI_FREE_MEM(BASE, IERROR)
[type] BASE(*) INTEGER IERROR
void MPI::Free_mem(void *base)

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

Rationale. The C and C++ bindings of MPI_ALLOC_MEM and MPI_FREE_MEM are similar to the bindings for the malloc and free C library calls: a call to MPI_Alloc_mem(...) &base should be paired with a call to MPI_Free_mem(base) (one less level of indirection). Both arguments are declared to be of same type void* so as to facilitate type casting. The Fortran binding is consistent with the C and C++ bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the (integer valued) address of the allocated memory. The base argument of MPI_FREE_MEM is a choice argument, which passes (a reference to) the variable stored at that location. (End of rationale.)

Advice to implementors. If MPI_ALLOC_MEM allocates special memory, then a design similar to the design of C malloc and free functions has to be used, in order to find out the size of a memory segment, when the segment is freed. If no special memory is used, MPI_ALLOC_MEM simply invokes malloc, and MPI_FREE_MEM invokes free.

A call to MPI_ALLOC_MEM can be used in shared memory systems to allocate memory in a shared memory segment. (End of advice to implementors.)

Example 8.1 Example of use of MPI_ALLOC_MEM, in Fortran with pointer support. We assume 4-byte REALS, and assume that pointers are address-sized.

```fortran
REAL A
POINTER (P, A(100,100)) ! no memory is allocated
CALL MPI_ALLOC_MEM(4*100*100, MPI_INFO_NULL, P, IERR)
! memory is allocated

A(3,5) = 2.71;

CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```
Since standard Fortran does not support (C-like) pointers, this code is not Fortran 77 or Fortran 90 code. Some compilers (in particular, at the time of writing, g77 and Fortran compilers for Intel) do not support this code.

Example 8.2 Same example, in C

```c
float (* f)[100][100]; /* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */

... (*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an MPI exception.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors will be handled should be read as may be handled.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_WORLD. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

- **MPI_ERRORS_ARE_FATAL** The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.

- **MPI_ERRORS_RETURN** The handler has no effect other than returning the error code to the user.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM_WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non-trivial MPI error handler.

Example 8.2 Same example, in C

```c
float (* f)[100][100]; /* no memory is allocated */
MPI_Alloc_mem(sizeof(float)*100*100, MPI_INFO_NULL, &f);
/* memory allocated */

... (*f)[5][3] = 2.71;
...
MPI_Free_mem(f);
```

8.3 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an MPI exception.

The above text takes precedence over any text on error handling within this document. Specifically, text that states that errors will be handled should be read as may be handled.

A user can associate error handlers to three types of objects: communicators, windows, and files. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for the respective object. MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_WORLD. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

- **MPI_ERRORS_ARE_FATAL** The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI_ABORT was called by the process that invoked the handler.

- **MPI_ERRORS_RETURN** The handler has no effect other than returning the error code to the user.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM_WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a non-trivial MPI error handler.
After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does not necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

Advice to implementors. A good quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C and C++ have distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

An error handler object is created by a call to MPI::XXX_CREATE_ERRHANDLER(function, erhandler), where XXX is, respectively, COMM, WIN, or FILE.

An error handler is attached to a communicator, window, or file by a call to MPI::XXX_SET_ERRHANDLER. The error handler must be either a predefined error handler, or an error handler that was created by a call to MPI::XXX_CREATE_ERRHANDLER, with matching XXX. The predefined error handlers MPI_ERRORS_RETURN and MPI_ERRORS_ARE_FATAL can be attached to communicators, windows, and files. In C+++, the predefined error handler MPI_ERRORS_THROW_EXCEPTIONS can also be attached to communicators, windows, and files.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI::XXX_GET_ERRHANDLER.

The function MPI::ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI::XXX_CREATE_ERRHANDLER.

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.

Advice to implementors. High-quality implementation should raise an error when an error handler that was created by a call to MPI::XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI::YYY_SET_ERRHANDLER.

To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (End of advice to implementors.)

The syntax for these calls is given below.

### 8.3.1 Error Handlers for Communicators
Chapter 8. MPI Environmental Management

MPI_ERRHANDLER function

1. **IN** function: user defined error handling procedure (function)
2. **OUT** errhandler: MPI error handler (handle)

The function `MPI_Comm_create_errhandler` can be used to create an error handler for a communicator. This function is identical to `MPI_ERRHANDLER_CREATE`, whose use is deprecated.

**Function Signature**

```c
int MPI_Comm_create_errhandler(MPI_Comm_errhandler_fn *function,
                                MPI_Errhandler *errhandler)
```

**Parameters**
- `function`: User-defined error handling procedure
- `errhandler`: MPI error handler

**Return Value**
- `0`: Success
- `< 0`: Failure

**Usage Notes**
- In C, the user routine should be a function of type `MPI_Comm_errhandler_fn`, defined as
  ```c
  typedef void MPI_Comm_errhandler_fn(MPI_Comm *, int *, ...);
  ``

- In Fortran, the user routine should be of the form:
  ```fortran
  SUBROUTINE COMM_ERRHANDLER_FN(COMM, ERROR_CODE, ...) 
  INTEGER COMM, ERROR_CODE
  
  Calls to COMM_ERRHANDLER_FN are discouraged from using a Fortran 
  {COMM|WIN|FILE}_ERRHANDLER_FN since the routine expects a variable number 
  of arguments. Some Fortran systems may allow this but some may fail to give the 
  correct result or compile/link this code. Thus, it will not, in general, be possible to 
  create portable code with a Fortran {COMM|WIN|FILE}_ERRHANDLER_FN. (End of 
  advice to users.)
  ``

- In C++, the user routine should be of the form:
  ```cpp
  typedef void MPI::Comm::Errhandler_fn(MPI::Comm & int *, ...);
  ``

**Rationale**
- The variable argument list is provided because it provides an ISO-standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments.

**Advice to users**
- A newly created communicator inherits the error handler that is associated with the “parent” communicator. In particular, the user can specify
8.3. ERROR HANDLING

a “global” error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (*End of advice to users.*)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)

INOUT comm communicator (handle)
IN errhandler new error handler for communicator (handle)

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)

INTEGER COMM, ERRHANDLER, IERROR

void MPI::Comm::Set_errhandler(const MPI::Errhandler&errhandler)

Attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_COMM_CREATE_ERRHANDLER. This call is identical to MPI_ERRHANDLER_SET, whose use is deprecated.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

IN comm communicator (handle)
OUT errhandler error handler currently associated with communicator (handle)

int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)

MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)

INTEGER COMM, ERRHANDLER, IERROR

MPI::Errhandler MPI::Comm::Get_errhandler() const

Retrieves the error handler currently associated with a communicator. This call is identical to MPI_ERRHANDLER_GET, whose use is deprecated.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

8.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(function, errhandler)

IN function user defined error handling procedure (function)
OUT errhandler MPI error handler (handle)

a “global” error handler for all communicators by associating this handler with the communicator MPI_COMM_WORLD immediately after initialization. (*End of advice to users.*)

MPI_COMM_SET_ERRHANDLER(comm, errhandler)

INOUT comm communicator (handle)
IN errhandler new error handler for communicator (handle)

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)

INTEGER COMM, ERRHANDLER, IERROR

void MPI::Comm::Set_errhandler(const MPI::Errhandler&errhandler)

Attaches a new error handler to a communicator. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_COMM_CREATE_ERRHANDLER. This call is identical to MPI_ERRHANDLER_SET, whose use is deprecated.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

IN comm communicator (handle)
OUT errhandler error handler currently associated with communicator (handle)

int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)

MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)

INTEGER COMM, ERRHANDLER, IERROR

MPI::Errhandler MPI::Comm::Get_errhandler() const

Retrieves the error handler currently associated with a communicator. This call is identical to MPI_ERRHANDLER_GET, whose use is deprecated.

Example: A library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

8.3.2 Error Handlers for Windows

MPI_WIN_CREATE_ERRHANDLER(function, errhandler)

IN function user defined error handling procedure (function)
OUT errhandler MPI error handler (handle)
int MPI_Win_create_errhandler(MPI_Win_errhandler_fn *function, 
    MPI_Errhandler *errhandler)

MPI_WIN_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)

EXTERNAL FUNCTION
INTEGER ERRHANDLER, IERROR

static MPI::Errhandler MPI::Win::Create_errhandler(MPI::Win::Errhandler_fn* 
    function)

Creates an error handler that can be attached to a window object. The user routine 
should be, in C, a function of type MPI_Win_errhandler_fn, which is defined as 
typedef void MPI_Win_errhandler_fn(MPI_Win *, int *, ...);

The first argument is the window in use, the second is the error code to be returned. 
In Fortran, the user routine should be of the form:
SUBROUTINE WIN_ERRHANDLER_FN(WIN, ERROR_CODE, ...) 
INTEGER WIN, ERROR_CODE

In C++, the user routine should be of the form:
typedef void MPI::Win::Errhandler_fn(MPI::Win &, int *, ...);

MPI_WIN_SET_ERRHANDLER(win, errhandler)

INOUT win window (handle)
IN errhandler new error handler for window (handle)

int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)

MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)

INTEGER WIN, ERRHANDLER, IERROR

void MPI::Win::Set_errhandler(const MPI::Errhandler& errhandler)

Attaches a new error handler to a window. The error handler must be either a pre-
defined error handler, or an error handler created by a call to 
MPI_WIN_CREATE_ERRHANDLER.

MPI_WIN_GET_ERRHANDLER(win, errhandler)

IN win window (handle)
OUT errhandler error handler currently associated with window (hand-
le)

int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)

MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)

INTEGER WIN, ERRHANDLER, IERROR

MPI::Errhandler MPI::Win::Get_errhandler() const
8.3. ERROR HANDLING

Retrieves the error handler currently associated with a window.

8.3.3 Error Handlers for Files

MPI_FILE_CREATE_ERRHANDLER(function, errhandler)

IN function user defined error handling procedure (function)

OUT errhandler MPI error handler (handle)

int MPI_File_create_errhandler(MPI_File_errhandler_fn *function, MPI_Errhandler *errhandler)

MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)

EXTERNAL FUNCTION INTEGER ERRHANDLER, IERROR

static MPI::Errhandler

MPI::File::Create_errhandler(MPI::File::Errhandler_fn* function)

Creates an error handler that can be attached to a file object. The user routine should be, in C, a function of type MPI_File_errhandler_fn, which is defined as

typedef void MPI_File_errhandler_fn(MPI_File *, int *, ...);

The first argument is the file in use, the second is the error code to be returned.

In Fortran, the user routine should be of the form:

SUBROUTINE FILE_ERRHANDLER_FN(FILE, ERROR_CODE, ...)

INTEGER FILE, ERROR_CODE

In C++, the user routine should be of the form:

typedef void MPI::File::Errhandler_fn(MPI::File &, int *, ...);

MPI_FILE_SET_ERRHANDLER(file, errhandler)

INOUT file file (handle)

IN errhandler new error handler for file (handle)

int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)

MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)

INTEGER FILE, ERRHANDLER, IERROR

void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler)

Attaches a new error handler to a file. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.

Retrieves the error handler currently associated with a window.

8.3.3 Error Handlers for Files

MPI_FILE_CREATE_ERRHANDLER(function, errhandler)

IN function user defined error handling procedure (function)

OUT errhandler MPI error handler (handle)

int MPI_File_create_errhandler(MPI_File_errhandler_fn *function, MPI_Errhandler *errhandler)

MPI_FILE_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)

EXTERNAL FUNCTION INTEGER ERRHANDLER, IERROR

static MPI::Errhandler

MPI::File::Create_errhandler(MPI::File::Errhandler_fn* function)

Creates an error handler that can be attached to a file object. The user routine should be, in C, a function of type MPI_File_errhandler_fn, which is defined as

typedef void MPI_File_errhandler_fn(MPI_File *, int *, ...);

The first argument is the file in use, the second is the error code to be returned.

In Fortran, the user routine should be of the form:

SUBROUTINE FILE_ERRHANDLER_FN(FILE, ERROR_CODE, ...)

INTEGER FILE, ERROR_CODE

In C++, the user routine should be of the form:

typedef void MPI::File::Errhandler_fn(MPI::File &, int *, ...);

MPI_FILE_SET_ERRHANDLER(file, errhandler)

INOUT file file (handle)

IN errhandler new error handler for file (handle)

int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)

MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)

INTEGER FILE, ERRHANDLER, IERROR

void MPI::File::Set_errhandler(const MPI::Errhandler& errhandler)

Attaches a new error handler to a file. The error handler must be either a predefined error handler, or an error handler created by a call to MPI_FILE_CREATE_ERRHANDLER.
CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_FILE_GET_ERRHANDLER(file, errhandler)
IN file file (handle)
OUT errhandler error handler currently associated with file (handle)

int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)

MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
INTEGER FILE, ERRHANDLER, IERROR
MPI::Errhandler MPI::File::Get_errhandler() const

Retrieves the error handler currently associated with a file.

8.3.4 Freeing Errorhandlers and Retrieving Error Strings

MPI_ERRHANDLER_FREE( errhandler )
INOUT errhandler MPI error handler (handle)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
INTEGER ERRHANDLER, IERROR
void MPI::Errhandler::Free()

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects associated with it (communicator, window, or file) have been deallocated.

MPI_ERROR_STRING( errorcode, string, resultlen )
IN errorcode Error code returned by an MPI routine
OUT string Text that corresponds to the errorcode
OUT resultlen Length (in printable characters) of the result returned in string

int MPI_Error_string(int errorcode, char *string, int *resultlen)

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
INTEGER ERRORCODE, RESULTLEN, IERROR
CHARACTER(*) STRING
void MPI::Get_error_string(int errorcode, char* name, int& resultlen)

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.
8.4. ERROR CODES AND CLASSES

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER(*) can not be referenced in, for example, a PRINT statement. (End of rationale.)

8.4 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called error classes. Valid error classes are shown in Table 8.1 and Table 8.2.

The error classes are a subset of the error codes: an MPI function may return an error class number; and the function MPI_ERROR_STRING can be used to compute the error string associated with an error class. An MPI error class is a valid MPI error code. Specifically, the values defined for MPI error classes are valid MPI error codes.

The error codes satisfy,

\[ 0 = \text{MPI_SUCCESS} < \text{MPI_ERR} \ldots \leq \text{MPI_ERR_LASTCODE}. \]

Rationale. The difference between MPI_ERR_UNKNOWN and MPI_ERR_OTHER is that MPI_ERROR_STRING can return useful information about MPI_ERR_OTHER.

Note that MPI_SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. (End of rationale.)

MPI_ERROR_CLASS(errorcode, errorclass)
IN  errorcode  Error code returned by an MPI routine
OUT errorclass  Error class associated with errorcode

int MPI_Error_class(int errorcode, int *errorclass)

MPI_ERROR_CLASS(errorcode, errorclass, IERROR)
INTEGER errorcode, errorclass, IERROR

int MPI::Get_error_class(int errorcode)

The function MPI_ERROR_CLASS maps each standard error code (error class) onto itself.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER(*) can not be referenced in, for example, a PRINT statement. (End of rationale.)

MPI_ERROR_CLASS(errorcode, errorclass)
IN  errorcode  Error code returned by an MPI routine
OUT errorclass  Error class associated with errorcode

int MPI_Error_class(int errorcode, int *errorclass)

MPI_ERROR_CLASS(errorcode, errorclass, IERROR)
INTEGER errorcode, errorclass, IERROR

int MPI::Get_error_class(int errorcode)

The function MPI_ERROR_CLASS maps each standard error code (error class) onto itself.
Table 8.1: Error classes (Part 1)

<table>
<thead>
<tr>
<th>Error Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>No error.</td>
</tr>
<tr>
<td>MPI_ERR_BUFFER</td>
<td>Invalid buffer pointer.</td>
</tr>
<tr>
<td>MPI_ERR_COUNT</td>
<td>Invalid count argument.</td>
</tr>
<tr>
<td>MPI_ERR_TYPE</td>
<td>Invalid datatype argument.</td>
</tr>
<tr>
<td>MPI_ERR_TAG</td>
<td>Invalid tag argument.</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
<td>Invalid communicator.</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
<td>Invalid rank.</td>
</tr>
<tr>
<td>MPI_ERR_REQUEST</td>
<td>Invalid request (handle).</td>
</tr>
<tr>
<td>MPI_ERR_ROOT</td>
<td>Invalid root.</td>
</tr>
<tr>
<td>MPI_ERR_GROUP</td>
<td>Invalid group.</td>
</tr>
<tr>
<td>MPI_ERR_OP</td>
<td>Invalid operation.</td>
</tr>
<tr>
<td>MPI_ERR_TOPOLOGY</td>
<td>Invalid topology.</td>
</tr>
<tr>
<td>MPI_ERR_DIMS</td>
<td>Invalid dimension argument.</td>
</tr>
<tr>
<td>MPI_ERR_ARG</td>
<td>Invalid argument of some other kind.</td>
</tr>
<tr>
<td>MPI_ERR_UNKNOWN</td>
<td>Unknown error.</td>
</tr>
<tr>
<td>MPI_ERR_TRUNCATE</td>
<td>Message truncated on receive.</td>
</tr>
<tr>
<td>MPI_ERR_OTHER</td>
<td>Known error not in this list.</td>
</tr>
<tr>
<td>MPI_ERR_IN_STATUS</td>
<td>Error code is in status.</td>
</tr>
<tr>
<td>MPI_ERR_PENDING</td>
<td>Pending request.</td>
</tr>
<tr>
<td>MPI_ERR_KEYVAL</td>
<td>Invalid keyval has been passed.</td>
</tr>
<tr>
<td>MPI_ERR_NOPMEM</td>
<td>MPI_ALLOC_MEM failed because memory is exhausted.</td>
</tr>
<tr>
<td>MPI_ERR_BASE</td>
<td>Invalid base passed to MPI_FREE_MEM.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_KEY</td>
<td>Key longer than MPI_MAX_INFO_KEY.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_VALUE</td>
<td>Value longer than MPI_MAX_INFO_VALUE.</td>
</tr>
<tr>
<td>MPI_ERR_INFO_NOKEY</td>
<td>Invalid key passed to MPI_INFO_DELETE.</td>
</tr>
<tr>
<td>MPI_ERR_SPAWN</td>
<td>Error in spawning processes.</td>
</tr>
<tr>
<td>MPI_ERR_PORT</td>
<td>Invalid port name passed to MPI_COMM_CONNECT.</td>
</tr>
<tr>
<td>MPI_ERR_SERVICE</td>
<td>Invalid service name passed to MPI_UNPUBLISH_NAME.</td>
</tr>
<tr>
<td>MPI_ERR_NAME</td>
<td>Invalid service name passed to MPI_LOOKUP_NAME.</td>
</tr>
<tr>
<td>MPI_ERR_WIN</td>
<td>Invalid win 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_INFO</td>
<td>Invalid info 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>
8.5 Error Classes, Error Codes, and Error Handlers

Users may want to write a layered library on top of an existing MPI implementation, and this library may have its own set of error codes and classes. An example of such a library is an I/O library based on MPI, see Chapter 13 on page 373. For this purpose, functions are needed to:

1. add a new error class to the ones an MPI implementation already knows.
2. associate error codes with this error class, so that MPI_ERROR_CLASS works.
3. associate strings with these error codes, so that MPI_ERROR_STRING works.
4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes: it is not expected that an application will generate them in significant numbers.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERR_FILE</td>
<td>Invalid file handle</td>
</tr>
<tr>
<td>MPI_ERR_NOTSAME</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</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_DATAREP</td>
<td>Unsupported datarep passed to</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</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>
<tr>
<td>MPI_ERR_LASTCODE</td>
<td>Last error code</td>
</tr>
</tbody>
</table>

Table 8.2: Error classes (Part 2)
CHAPTER 8. MPI ENVIRONMENTAL MANAGEMENT

MPI_ADD_ERROR_CLASS(errorclass)

OUT errorclass value for the new error class (integer)

int MPI_Add_error_class(int *errorclass)

MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)

INTEGER ERRORCLASS, IERROR

int MPI::Add_error_class()

Creates a new error class and returns the value for it.

Rationale. To avoid conflicts with existing error codes and classes, the value is set by the implementation and not by the user. (End of rationale.)

Advice to implementors. A high-quality implementation will return the value for a new error class in the same deterministic way on all processes. (End of advice to implementors.)

Advice to users. Since a call to MPI_ADD_ERROR_CLASS is local, the same errorclass may not be returned on all processes that make this call. Thus, it is not safe to assume that registering a new error on a set of processes at the same time will yield the same errorclass on all of the processes. However, if an implementation returns the new errorclass in a deterministic way, and they are always generated in the same order on the same set of processes (for example, all processes), then the value will be the same. However, even if a deterministic algorithm is used, the value can vary across processes. This can happen, for example, if different but overlapping groups of processes make a series of calls. As a result of these issues, getting the “same” error on multiple processes may not cause the same value of error code to be generated. (End of advice to users.)

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-defined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. (End of advice to users.)

The value of MPI_ERR_LASTCODE is a constant value and is not affected by new user-defined error codes and classes. Instead, a predefined attribute key MPI_LASTUSEDCODE is associated with MPI_COMM_WORLD. The attribute value corresponding to this key is the current maximum error class including the user-defined ones. This is a local value and may be different on different processes. The value returned by this key is always greater than or equal to MPI_ERR_LASTCODE.

Advice to users. The value returned by the key MPI_LASTUSEDCODE will not change unless the user calls a function to explicitly add an error class/code. In a multi-threaded environment, the user must take extra care in assuming this value has not changed. Note that error codes and error classes are not necessarily dense. A user may not assume that each error class below MPI_LASTUSEDCODE is valid. (End of advice to users.)
8.5. ERROR CLASSES, ERROR CODES, AND ERROR HANDLERS

MPI_ADD_ERROR_CODE(errorclass, errorcode)
IN errorclass error class (integer)
OUT errorcode new error code to associated with errorclass (integer)

int MPI_Add_error_code(int errorclass, int *errorcode)

MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
INTEGER ERRORCLASS, ERRORCODE, IERROR

int MPI::::Add_error_code(int errorclass)

Creates new error code associated with errorclass and returns its value in errorcode.

Rationale. To avoid conflicts with existing error codes and classes, the value of the
new error code is set by the implementation and not by the user. (End of rationale.)

Advice to implementors. A high-quality implementation will return the value for
a new errorcode in the same deterministic way on all processes. (End of advice to
implementors.)

MPI_ADD_ERROR_STRING(errorcode, string)
IN errorcode error code or class (integer)
IN string text corresponding to errorcode (string)

int MPI_Add_error_string(int errorcode, char *string)

MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
INTEGER ERRORCODE, IERROR
CHARACTER*(*) STRING

void MPI::::Add_error_string(int errorcode, const char* string)

Associates an error string with an error code or class. The string must be no more
than MPI_MAX_ERROR_STRING characters long. The length of the string is defined in
the calling language. The length of the string does not include the null terminator in C
or C++. Trailing blanks will be stripped in Fortran. Calling MPI_ADD_ERROR_STRING
for an errorcode that already has a string will replace the old string with the new string.
It is erroneous to call MPI_ADD_ERROR_STRING for an error code or class with a value
\leq MPI_ERR_LASTCODE.

If MPI_ERROR_STRING is called when no string has been set, it will return a empty
string (all spaces in Fortran, ** in C and C++).

Section 8.3 on page 264 describes the methods for creating and associating error han-
dlers with communicators, files, and windows.
This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. Users should note that the default error handler is MPI_ERRORS_ARE_FATAL. Thus, calling MPI_COMM_CALL_ERRHANDLER will abort the comm processes if the default error handler has not been changed for this communicator or on the parent before the communicator was created. (End of advice to users.)

MPI_Win::Call_errhandler(int errorcode) const

This function invokes the error handler assigned to the window with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. As with communicators, the default error handler for windows is MPI_ERRORS_ARE_FATAL. (End of advice to users.)

MPI_File::Call_errhandler(fh, errorcode)

IN fh file with error handler (handle)
IN errorcode error code (integer)

This function invokes the error handler assigned to the file with the error code supplied. This function returns MPI_SUCCESS in C and C++ and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

Advice to users. As with communicators, the default error handler for windows is MPI_ERRORS_ARE_FATAL. (End of advice to users.)


```c
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
double MPI_Wtime(void)
```

8.6. TIMERS AND SYNCHRONIZATION

MPI defines a timer. A timer is specified even though it is not “message-passing,” because timing parallel programs is important in “performance debugging” and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers. See also Section 2.6.5 on page 21.

8.6.1 Timing Parallel Programs

Advice to users. Users are warned that handlers should not be called recursively with MPI_WTIME, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_WTIME, MPI_FILE_CALL_ERRHANDLER, or MPI_WIN_CALL_ERRHANDLER is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (End of advice to users.)

8.6 Timers and Synchronization

MPI defines a timer. A timer is specified even though it is not “message-passing,” because timing parallel programs is important in “performance debugging” and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers. See also Section 2.6.5 on page 21.

Advice to users. Unlike errors on communicators and windows, the default behavior for files is to have MPI_ERRORS_RETURN. (End of advice to users.)

Advice to users. Users are warned that handlers should not be called recursively with MPI_Comm_call_errhandler, MPI_File_call_errhandler, or MPI_Win_call_errhandler. Doing this can create a situation where an infinite recursion is created. This can occur if MPI_Comm_call_errhandler, MPI_File_call_errhandler, or MPI_Win_call_errhandler is called inside an error handler.

Error codes and classes are associated with a process. As a result, they may be used in any error handler. Error handlers should be prepared to deal with any error code they are given. Furthermore, it is good practice to only call an error handler with the appropriate error codes. For example, file errors would normally be sent to the file error handler. (End of advice to users.)
This function is portable (it returns seconds, not “ticks”), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```c
{ double starttime, endtime;
  starttime = MPI_Wtime();
  .... stuff to be timed ....
  endtime = MPI_Wtime();
  printf("That took %f seconds\n", endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return “the same time.” (But see also the discussion of MPI_WTIME_IS_GLOBAL).

`MPI_WTICK()`

```c
double MPI_Wtick(void)
DOUBLE PRECISION MPI_WTICK()
double MPI::Wtick()
```

`MPI_WTICK()` returns the resolution of `MPI_WTIME` in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by `MPI_WTICK` should be $10^{-3}$.

### 8.7 Startup

One goal of MPI is to achieve source code portability. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does not say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialization routine `MPI_INIT`.

`MPI_INIT()`

```c
int MPI_Init(int *argc, char ***argv)
MPI_INIT(IERROR)
INTEGER IERROR
void MPI::Init(int& argc, char**& argv)
void MPI::Init()
```
This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see MPI\_INITIALIZED).

All MPI programs must contain a call to MPI\_INIT; this routine must be called before any other MPI routine (apart from MPI\_GET\_VERSION, MPI\_INITIALIZED, and MPI\_FINALIZED) is called. The version for ISO C accepts the argc and argv that are provided by the arguments to main:

```c
int main(argc, argv)
```

```c
int argc;
char *argv;
{
  MPI\_Init(&argc, &argv);
  /* parse arguments */
  /* main program */
  MPI\_Finalize(); /* see below */
}
```

The Fortran version takes only IERROR. Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C and C++. In C++, there is an alternative binding for MPI\_Init that does not have these arguments at all.

Rationale: In some applications, libraries may be making the call to MPI\_Init, and may not have access to argc and argv from main. It is anticipated that applications requiring special information about the environment or information supplied by mpiexec can get that information from environment variables. (End of rationale.)

**MPI\_FINALIZE()**

```c
int MPI\_Finalize(void)
MPI\_FINALIZE(IERROR)
INTEGER IERROR
void MPI::\_Finalize()
```

This routine cleans up all MPI state. Each process must call MPI\_FINALIZE before it exits. Unless there has been a call to MPI\_ABORT, each process must ensure that all pending non-blocking communications are (locally) complete before calling MPI\_FINALIZE.

Further, at the instant at which the last process calls MPI\_FINALIZE, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

For example, the following program is correct:

```
Process 0     Process 1
-----        ------
MPI\_Init();  MPI\_Init();
```

This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous (see MPI\_INITIALIZED).

All MPI programs must contain a call to MPI\_INIT; this routine must be called before any other MPI routine (apart from MPI\_GET\_VERSION, MPI\_INITIALIZED, and MPI\_FINALIZED) is called. The version for ISO C accepts the argc and argv that are provided by the arguments to main:

```c
int main(argc, argv)
```

```c
int argc;
char *argv;
{
  MPI\_Init(&argc, &argv);
  /* parse arguments */
  /* main program */
  MPI\_Finalize(); /* see below */
}
```

The Fortran version takes only IERROR. Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C and C++. In C++, there is an alternative binding for MPI\_Init that does not have these arguments at all.

Rationale: In some applications, libraries may be making the call to MPI\_Init, and may not have access to argc and argv from main. It is anticipated that applications requiring special information about the environment or information supplied by mpiexec can get that information from environment variables. (End of rationale.)

**MPI\_FINALIZE()**

```c
int MPI\_Finalize(void)
MPI\_FINALIZE(IERROR)
INTEGER IERROR
void MPI::\_Finalize()
```

This routine cleans up all MPI state. Each process must call MPI\_FINALIZE before it exits. Unless there has been a call to MPI\_ABORT, each process must ensure that all pending non-blocking communications are (locally) complete before calling MPI\_FINALIZE.

Further, at the instant at which the last process calls MPI\_FINALIZE, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

For example, the following program is correct:

```
Process 0     Process 1
-----        ------
MPI\_Init();  MPI\_Init();
```
Without the matching receive, the program is erroneous:

A successful return from a blocking communication operation or from `MPI_FINALIZE` tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from `MPI_REQUEST_FREE` with a request handle generated by an `MPI_ISSEND` nullifies the handle but provides no assurance of operation completion. The `MPI_FINALIZE` guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

`MPI_FINALIZE` guarantees nothing about pending communications that have not been completed (completion is assured only by `MPI_WAIT`, `MPI_TEST`, or `MPI_REQUEST_FREE` combined with some other verification of completion).

**Example 8.3** This program is correct:

```c
MPI_Init();
MPI_Send(dest=1);
MPI_Finalize();
```

```c
MPI_Init();
MPI_Recv(src=0);
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Send(dest=1);
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv(src=0);
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.4** This program is erroneous and its behavior is undefined:

```c
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
exit();
```

```c
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

**Example 8.5** This program is correct, and after the `MPI_Finalize`, it is as if the buffer had been detached.

```c
MPI_Init();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.5** This program is correct, and after the `MPI_Finalize`, it is as if the buffer had been detached.

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

A successful return from a blocking communication operation or from `MPI_FINALIZE` tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from `MPI_REQUEST_FREE` with a request handle generated by an `MPI_ISSEND` nullifies the handle but provides no assurance of operation completion. The `MPI_FINALIZE` is complete only when it is known by some means that a matching receive has completed. `MPI_FINALIZE` guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

`MPI_FINALIZE` guarantees nothing about pending communications that have not been completed (completion is assured only by `MPI_WAIT`, `MPI_TEST`, or `MPI_REQUEST_FREE` combined with some other verification of completion).

**Example 8.3** This program is correct:

```c
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
exit();
```

```c
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.4** This program is erroneous and its behavior is undefined:

```c
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
exit();
```

```c
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.5** This program is correct, and after the `MPI_Finalize`, it is as if the buffer had been detached.

```c
MPI_Init();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

A successful return from a blocking communication operation or from `MPI_FINALIZE` tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from `MPI_REQUEST_FREE` with a request handle generated by an `MPI_ISSEND` nullifies the handle but provides no assurance of operation completion. The `MPI_FINALIZE` is complete only when it is known by some means that a matching receive has completed. `MPI_FINALIZE` guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

`MPI_FINALIZE` guarantees nothing about pending communications that have not been completed (completion is assured only by `MPI_WAIT`, `MPI_TEST`, or `MPI_REQUEST_FREE` combined with some other verification of completion).

**Example 8.3** This program is correct:

```c
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```c
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.4** This program is erroneous and its behavior is undefined:

```c
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
exit();
```

```c
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.5** This program is correct, and after the `MPI_Finalize`, it is as if the buffer had been detached.

```c
MPI_Init();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

A successful return from a blocking communication operation or from `MPI_FINALIZE` tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from `MPI_REQUEST_FREE` with a request handle generated by an `MPI_ISSEND` nullifies the handle but provides no assurance of operation completion. The `MPI_FINALIZE` is complete only when it is known by some means that a matching receive has completed. `MPI_FINALIZE` guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

`MPI_FINALIZE` guarantees nothing about pending communications that have not been completed (completion is assured only by `MPI_WAIT`, `MPI_TEST`, or `MPI_REQUEST_FREE` combined with some other verification of completion).

**Example 8.3** This program is correct:

```c
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
exit();
```

```c
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.4** This program is erroneous and its behavior is undefined:

```c
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
exit();
```

```c
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Isend();
MPI_Request_free();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Recv();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```

**Example 8.5** This program is correct, and after the `MPI_Finalize`, it is as if the buffer had been detached.

```c
MPI_Init();
MPI_Finalize();
```

```
Process 0
-----------
MPI_Init();
MPI_Finalize();
```

```
Process 1
-----------
MPI_Init();
MPI_Finalize();
```

```
MPI_Send(dest=1); MPI_Recv(src=0); MPI_Finalize(); MPI_Finalize();
```
8.7. STARTUP

rank 0

---

buffer = malloc(1000000);
MPI_Recv();
MPI_Barrier();
MPI_Cancel();
MPI_Finalize();
free(buffer);
exit();

Example 8.6 In this example, MPI_Iprobe() must return a FALSE flag.

Advice to implementors. An implementation may need to delay the return from

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except
for MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. Each process must
complete any pending communication it initiated before it calls MPI_FINALIZE. If the call
returns, each process may continue local computations, or exit, without participating in
further MPI communication with other processes. MPI_FINALIZE is collective over all con-
nected 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.

Advice to implementors. Even though a process has completed all the communication
it initiated, such communication may not yet be completed from the viewpoint of the

8.7. STARTUP

rank 1

---

buffer = malloc(1000000);
MPI_Recv();
MPI_Finalize();
free(buffer);
exit();

Example 8.6 In this example, MPI_Iprobe() must return a FALSE flag.

Advice to implementors. An implementation may need to delay the return from

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except
for MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. Each process must
complete any pending communication it initiated before it calls MPI_FINALIZE. If the call
returns, each process may continue local computations, or exit, without participating in
further MPI communication with other processes. MPI_FINALIZE is collective over all con-
nected 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.

Advice to implementors. Even though a process has completed all the communication
it initiated, such communication may not yet be completed from the viewpoint of the
underlying MPI system. E.g., a blocking send may have completed, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. (End of advice to implementors.)

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, they may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 8.7 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```c
...MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile","w");
    dump_results(resultfile);
    fclose(resultfile);
}exit(0);
```

**MPI_INITIALIZED( flag )**

Out flag

```c
int MPI_Initialized(int *flag)
MPI_INITIALIZED(FLAG, IERROR)
LOGICAL FLAG
INTEGER IERROR
bool MPI::Is_initialized()
```

This routine may be used to determine whether MPI_INIT has been called. MPI_INITIALIZED returns true if the calling process has called MPI_INIT. Whether MPI_FINALIZE has been called does not affect the behavior of MPI_INITIALIZED. It is one of the few routines that may be called before MPI_INIT is called.
8.7. STARTUP

MPI_ABORT( comm, errorcode )

IN comm communicator of tasks to abort
IN errorcode error code to return to invoking environment

int MPI_ABORT(MPI_Comm comm, int errorcode)

MPI_ABORT(COMM, ERRORCODE, IERROR)

INTEGER COMM, ERRORCODE, IERROR

void MPI::Comm::Abort(int errorcode)

This routine makes a “best attempt” to abort all tasks in the group of comm. This function does not require that the invoking environment take any action with the error code. However, a Unix or POSIX environment should handle this as a return errorcode from the main program.

It may not be possible for an MPI implementation to abort only the processes represented by comm if this is a subset of the processes. In this case, the MPI implementation should attempt to abort all the connected processes but should not abort any unconnected processes. If no processes were spawned, accepted or connected then this has the effect of aborting all the processes associated with MPI_COMM_WORLD.

Rationale. The communicator argument is provided to allow for future extensions of MPI to environments with, for example, dynamic process management. In particular, it allows but does not require an MPI implementation to abort a subset of MPI_COMM_WORLD. (End of rationale.)

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 implementers. 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 implementers.)

8.7.1 Allowing User Functions at Process Termination

There are times in which it would be convenient to have actions happen when an MPI process finishes. For example, a routine may do initializations that are useful until the MPI job (or that part of the job that being terminated in the case of dynamically created processes) is finished. This can be accomplished in MPI by attaching an attribute to MPI_COMM_SELF with a callback function. When MPI_FINALIZE is called, it will first execute the equivalent of an MPI_COMM_FREE on MPI_COMM_SELF. This will cause the delete callback function to be executed on all keys associated with MPI_COMM_SELF, in an arbitrary order. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The “freeing” of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.
Advice to implementors. Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. (End of advice to implementors.)

8.7.2 Determining Whether MPI Has Finished

One of the goals of MPI was to allow for layered libraries. In order for a library to do this cleanly, it needs to know if MPI is active. In MPI the function MPI_INITIALIZED was provided to tell if MPI had been initialized. The problem arises in knowing if MPI has been finalized. Once MPI has been finalized it is no longer active and cannot be restarted. A library needs to be able to determine this to act accordingly. To achieve this the following function is needed:

```c
MPI_FINALIZED(flag)
```

OUT flag true if MPI was finalized (logical)

```c
int MPI_Finalized(int *flag)
```

MPI_FINALIZED(FLAG, IERROR)
LOGICAL FLAG
INTEGER IERROR

```c
bool MPI::is_finalized()
```

This routine returns true if MPI_FINALIZE has completed. It is legal to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE.

Advice to users. MPI is “active” and it is thus safe to call MPI functions if MPI_INIT has completed and MPI_FINALIZE has not completed. If a library has no other way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is “active” in callback functions that are invoked during MPI_FINALIZE. (End of advice to users.)

8.8 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

```
mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard startup mechanism. In order that the “standard” command not be confused with

Advice to users. MPI is “active” and it is thus safe to call MPI functions if MPI_INIT has completed and MPI_FINALIZE has not completed. If a library has no other way of knowing whether MPI is active or not, then it can use MPI_INITIALIZED and MPI_FINALIZED to determine this. For example, MPI is “active” in callback functions that are invoked during MPI_FINALIZE. (End of advice to users.)

8.8 Portable MPI Process Startup

A number of implementations of MPI provide a startup command for MPI programs that is of the form

```
mpirun <mpirun arguments> <program> <program arguments>
```

Separating the command to start the program from the program itself provides flexibility, particularly for network and heterogeneous implementations. For example, the startup script need not run on one of the machines that will be executing the program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard startup mechanism. In order that the “standard” command not be confused with
existing practice, which is not standard and not portable among implementations, instead of 

\texttt{mpirun} MPI specifies \texttt{mpiexec}.

While a standardized startup mechanism improves the usability of MPI, the range of 
environments is so diverse (e.g., there may not even be a command line interface) that MPI 
cannot mandate such a mechanism. Instead, MPI specifies an \texttt{mpiexec} startup command 
and recommends but does not require it, as advice to implementors. However, if an 
implementation does provide a command called \texttt{mpiexec}, it must be of the form described 
below.

It is suggested that

\texttt{mpiexec -n <numprocs> <program>}

be at least one way to start \texttt{<program>} with an initial MPI \texttt{COMM_WORLD} whose group 
contains \texttt{<numprocs>} processes. Other arguments to \texttt{mpiexec} may be implementation-
dependent.

**Advice to implementors.** Implementors, if they do provide a special startup command 
for MPI programs, are advised to give it the following form. The syntax is chosen in 
order that \texttt{mpiexec} be able to be viewed as a command-line version of 
\texttt{MPI_COMM_SPAWN} (See Section 10.3.4).

Analogous to \texttt{MPI_COMM_SPAWN}, we have

\begin{verbatim}
mpiexec -n <maxprocs>
-soft < >-host < >-arch < >-wdir < >-path < >-file < >
...
<command line>
\end{verbatim}

for the case where a single command line for the application program and its arguments 
will suffice. See Section 10.3.4 for the meanings of these arguments. For the case 
corresponding to \texttt{MPI_COMM_SPAWN_MULTIPLE} there are two possible formats:

Form A:

\begin{verbatim}
mpiexec ( <above arguments> ) : { ... } : { ... } : ... : { ... }
\end{verbatim}

As with \texttt{MPI_COMM_SPAWN}, all the arguments are optional. (Even the \texttt{-n} \texttt{x} 
argument is optional; the default is implementation dependent. It might be 1, it might be 
taken from an environment variable, or it might be specified at compile time.) The 
names and meanings of the arguments are taken from the keys in the \texttt{info} argument 
to \texttt{MPI_COMM_SPAWN}. There may be other, implementation-dependent arguments 
as well.

Note that Form A, though convenient to type, prevents colons from being program 
arguments. Therefore an alternate, file-based form is allowed:

Form B:

\begin{verbatim}
mpiexec -n <numprocs> <program>
\end{verbatim}

be at least one way to start \texttt{<program>} with an initial MPI \texttt{COMM_WORLD} whose group 
contains \texttt{<numprocs>} processes. Other arguments to \texttt{mpiexec} may be implementation-
dependent.

**Advice to implementors.** Implementors, if they do provide a special startup command 
for MPI programs, are advised to give it the following form. The syntax is chosen in 
order that \texttt{mpiexec} be able to be viewed as a command-line version of 
\texttt{MPI_COMM_SPAWN} (See Section 10.3.4).

Analogous to \texttt{MPI_COMM_SPAWN}, we have

\begin{verbatim}
mpiexec -n <maxprocs>
-soft < >-host < >-arch < >-wdir < >-path < >-file < >
...
<command line>
\end{verbatim}

for the case where a single command line for the application program and its arguments 
will suffice. See Section 10.3.4 for the meanings of these arguments. For the case 
corresponding to \texttt{MPI_COMM_SPAWN_MULTIPLE} there are two possible formats:

Form A:

\begin{verbatim}
mpiexec ( <above arguments> ) : { ... } : { ... } : ... : { ... }
\end{verbatim}

As with \texttt{MPI_COMM_SPAWN}, all the arguments are optional. (Even the \texttt{-n} \texttt{x} 
argument is optional; the default is implementation dependent. It might be 1, it might be 
taken from an environment variable, or it might be specified at compile time.) The 
names and meanings of the arguments are taken from the keys in the \texttt{info} argument 
to \texttt{MPI_COMM_SPAWN}. There may be other, implementation-dependent arguments 
as well.

Note that Form A, though convenient to type, prevents colons from being program 
arguments. Therefore an alternate, file-based form is allowed:

Form B:
mpiexec -configfile <filename>

where the lines of `<filename>` are of the form separated by the colons in Form A.
Lines beginning with `#` are comments, and lines may be continued by terminating
the partial line with `\`.

Example 8.8 Start 16 instances of `myprog` on the current or default machine:

\begin{verbatim}
mpiexec -n 16 myprog
\end{verbatim}

Example 8.9 Start 10 processes on the machine called `ferrari`:

\begin{verbatim}
mpiexec -n 10 -host ferrari myprog
\end{verbatim}

Example 8.10 Start three copies of the same program with different command-line arguments:

\begin{verbatim}
mpiexec myprog infile1 : myprog infile2 : myprog infile3
\end{verbatim}

Example 8.11 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000’s:

\begin{verbatim}
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
\end{verbatim}

It is assumed that the implementation in this case has a method for choosing hosts of
the appropriate type. Their ranks are in the order specified.

Example 8.12 Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000’s (Form B):

\begin{verbatim}
mpiexec -configfile myfile
\end{verbatim}

where `myfile` contains

\begin{verbatim}
-n 5 -arch sun ocean
-n 10 -arch rs6000 atmos
\end{verbatim}

(End of advice to implementors.)
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). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include the strings "true" and "false" (all lowercase). For integers, legal values must include
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

void MPI::Info::Set(const char* key, const char* value)

MPI_INFO_SET adds 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(info, key)
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)

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. valuelen 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)
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)

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. valuelen 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)
INTEGER INFO, NKEYS, IERROR

int MPI::Info::Get_nkeys() const
MPI_INFO_GET_NKEYS returns the number of currently defined keys in info.

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)
INTEGER INFO, N, IERROR

void MPI::Info::Get_nthkey(int n, char* key) const
This function returns the nth defined key in info. Keys are numbered 0 . . . N − 1 where N is the value returned by MPI_INFO_GET_NKEYS. All keys between 0 and N − 1 are guaranteed to be defined. The number of a given key does not change as long as info is not modified with MPI_INFO_SET or MPI_INFO_DELETE.

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)

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 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 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 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.
10.2. THE DYNAMIC PROCESS MODEL

- **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.)

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)

• 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.)

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)
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[]) const

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

MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program specified by command, establishing communication with them and returning an intercommunicator. The spawned processes are referred to as children. The children have their own MPI_COMM_WORLD, which is separate from that of the parents. MPI_COMM_SPAWN is collective over comm, and also may not return until MPI_INIT has been called in the children. Similarly, MPI_INIT in the children may not return until all parents have called MPI_COMM_SPAWN. In this sense, MPI_COMM_SPAWN in the parents and MPI_INIT in the children form a collective operation over the union of parent and child processes. The intercommunicator returned by MPI_COMM_SPAWN contains the parent processes in the local group and the child processes in the remote group. The ordering of processes in the local and remote groups is the same as the ordering of the group of the comm in the parents and of MPI_COMM_WORLD of the children, respectively. This intercommunicator can be obtained in the children through the function MPI_COMM_GET_PARENT.

Advice to users. An implementation may automatically establish communication before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN in the parent does not necessarily mean that MPI_INIT has been called in the children (although the returned intercommunicator can be used immediately). (End of advice to users.)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in $HOME/pvm3/bin/$PVM_ARCH). An MPI implementation running in the parent does not necessarily mean that MPI_INIT has been called in the children.

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in $HOME/pvm3/bin/$PVM_ARCH). An MPI implementation running in the parent does not necessarily mean that MPI_INIT has been called in the children.

Advice to users. An implementation may automatically establish communication before MPI_INIT is called by the children. Thus, completion of MPI_COMM_SPAWN in the parent does not necessarily mean that MPI_INIT has been called in the children (although the returned intercommunicator can be used immediately). (End of advice to users.)

The command argument The command argument is a string containing the name of a program to be spawned. The string is null-terminated in C. In Fortran, leading and trailing spaces are stripped. MPI does not specify how to find the executable or how the working directory is determined. These rules are implementation-dependent and should be appropriate for the runtime environment.

Advice to implementors. The implementation should use a natural rule for finding executables and determining working directories. For instance, a homogeneous system with a global file system might look first in the working directory of the spawning process, or might search the directories in a PATH environment variable as do Unix shells. An implementation on top of PVM would use PVM's rules for finding executables (usually in $HOME/pvm3/bin/$PVM_ARCH). An MPI implementation running in the parent does not necessarily mean that MPI_INIT has been called in the children.
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, ...)
```

and on 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, ...)
```
10.3. PROCESS MANAGER INTERFACE

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI_COMM_SPAWN argument argv differs from the argv argument of main in two respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the implementation and conventionally contains the name of the program (given by command). argv[1] of main corresponds to argv[0] in MPI_COMM_SPAWN, argv[2] of main to argv[1] of MPI_COMM_SPAWN, etc. 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 \{ni : 0 \leq ni \leq \text{maxprocs}\} of allowed values for the number of processes spawned. The set \{ni\} 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_errors 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 \(\{n_i\}\) is \(\{0 \ldots N\}\). However, this is not completely portable, as implementations are not required to support soft spawning. (End of advice to users.)

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 \( m \) \( (0 \leq m < \text{maxprocs}) \) processes were 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.)

```c
MPI_COMM_GET_PARENT(parent)
OUT parent the parent communicator (handle)
```

```fortran
integer :: mpi_comm_get_parent(MPI_Comm *parent)
MPI_COMM_GET_PARENT(PARENT, IERROR)
INTEGER PARENT, IERROR
```

```c++
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.

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 \( m \) \( (0 \leq m < \text{maxprocs}) \) processes were 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.)

```c
MPI_COMM_GET_PARENT(parent)
OUT parent the parent communicator (handle)
```

```fortran
integer :: mpi_comm_get_parent(MPI_Comm *parent)
MPI_COMM_GET_PARENT(PARENT, IERROR)
INTEGER PARENT, IERROR
```

```c++
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.
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 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 intracommmunicator 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 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 intracommmunicator 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.)
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
ARRAY_OF_ERRCODES, ERROR)

INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM,
INTERCOMM, ARRAY_OF_ERRCODES(*), ERROR

CHARACTER(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(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::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 MPI_COMM_SPAWN_MULTIPLE, the element array_of_argv(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 ***)0 in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array_of_argv to MPI_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_argv` 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];
charargv0[] = "-gridfile", "ocean1.grd", (char *)0;
charargv1[] = "atmos.grd", (char *)0;
array_of_argv[0] = 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(1, 2) = ' ocean1.grd'
array_of_argv(1, 3) = ' '
commands(2) = ' atmos'
array_of_argv(2, 1) = ' atmos.grd '
array_of_argv(2, 2) = ' '
array_of_argv(2, 3) = ' '

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.

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];
charargv0[] = "-gridfile", "ocean1.grd", (char *)0;
charargv1[] = "atmos.grd", (char *)0;
array_of_argv[0] = 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(1, 2) = ' ocean1.grd'
array_of_argv(1, 3) = ' '
commands(2) = ' atmos'
array_of_argv(2, 1) = ' atmos.grd '
array_of_argv(2, 2) = ' '
array_of_argv(2, 3) = ' '

call MPI_COMM_SPAWN_MULTIPLE(2, commands, array_of_argv, ...)
```
**wdir** 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.

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

**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:

- a:b gives a range between a and b
- 0:N gives full "soft" functionality
- 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows power-of-two number of processes.
- 2:10000:2 allows even number of processes.
- 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
#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_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm everyone; /* intercommunicator */
    char worker_program[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

Manager-worker Example, Using MPI_COMM_SPAWN.

```c
#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_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
         processes 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");

    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
             processes 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");
CHAPTER 10. PROCESS CREATION AND MANAGEMENT

/* Parallel code here. */
/* The manager is represented as the process with rank 0 in (the remote */
/* group of) the parent communicator. If the workers need to communicate */
/* among themselves, they can use MPI_COMM_WORLD. */

MPI_Finalize(); return 0;
}

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 resides at a well-known internet address host:port.
- The server prints out an address to the terminal, the user gives this address to the client program.
- The server places the address information on a nameserver, where it can be retrieved with an agreed-upon name.
- The server to which the client connects is actually a broker, acting as a middleman between the client and the real server.

MPI does not require a nameserver, so not all implementations will be able to support all of the above scenarios. However, MPI provides an optional nameserver interface, and is compatible with external name servers.

A port name is a system-supplied string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol. The server establishes a port name with the MPI_OPEN_PORT routine. It accepts a connection to a given port with MPI_COMM_ACCEPT. A client uses port name to connect to the server.

By itself, the port name mechanism is completely portable, but it may be clumsy to use because of the necessity to communicate port name to the client. It would be more convenient if a server could specify that it be known by an application-supplied service name so that the client could connect to that service name without knowing the port name.

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.
10.4.2 Server Routines

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)

void MPI::Open_port(const MPI::Info& info, char* port_name)

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.)

MPI_COMM_ACCEPT(info, port_name)
IN info implementation-specific information on how to establish an address (handle)
OUT port_name newly established port (string)

int MPI_Comm_accept(MPI_Info info, char *port_name)
MPI_COMM_ACCEPT(INFO, PORT_NAME, IERROR)

void MPI::Comm_accept(const MPI::Info& info, char* port_name)

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.)
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(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)

IN port_name network address (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 server as remote group (handle)

int MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)

CHARACTER(*) PORT_NAME
INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
MPI::Intercomm MPI::Intracomm::Connect(const char* port_name, const MPI::Info& info, int root) const

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 info arguments to MPI_OPEN_PORT, MPI_COMM_ACCEPT and/or MPI_COMM_CONNECT, for the user to control timeout and queuing behavior. (End of advice to implementors.)

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order they were initiated and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

port_name is the address of the server. It must be the same as the name returned by MPI_OPEN_PORT on the server. Some freedom is allowed here. If there are equivalent forms of port_name, an implementation may accept them as well. For instance, if port_name is (hostname:port), an implementation may accept (ip_address:port) as well.
10.4. Establishing Communication

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 argument 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(char *service_name, MPI_Info info, char *port_name)

MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
INTEGER INFO, IERROR
CHARACTER*(*), SERVICE_NAME, PORT_NAME

void MPI::Publish_name(const char*, service_name, const MPI::Info, const char*)

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 could easily conflict with a name published by a user. In order to avoid such conflicts,
MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

The following situation is problematic but unavoidable, if we want to allow implementations to use nameservers. Suppose there are multiple instances of “ocean” running on a machine. If the scope of a service name is confined to a job, then multiple oceans can coexist. If an implementation provides site-wide scope, however, multiple instances are not possible as all calls to MPI_PUBLISH_NAME after the first may fail. There is no universal solution to this.

To handle these situations, a high-quality implementation should make it possible to limit the domain over which names are published. (End of advice to implementors.)

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)

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

MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)

INTEGER INFO, IERROR
CHARACTER*(*) SERVICE_NAME, PORT_NAME

void MPI::Unpublish_name(const char* service_name, const MPI::Info& info, const char* port_name)

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)

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

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

MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
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.

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:

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
```

The server prints out the port name to the terminal and the user must type it in when starting up the client (assuming the MPI implementation supports stdin such that this works). On the client side:

```
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
```

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.

```
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.

10.4.6 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:

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
```

The server prints out the port name to the terminal and the user must type it in when starting up the client (assuming the MPI implementation supports stdin such that this works). On the client side:

```
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
```
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);

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_init(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) {
            /* do something */
            MPI_Get(&status, &client);
            again = 0;
            if (status.MPI_REQUEST == MPI_REQUEST_ABNORMAL)
                break;
            /* process data */
            MPI_Free(&status, &client);
        }
        break;
    }
    MPI_Finalize();
    return 0;
}
MPI_Recv( buf, MAX_DATA, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status );
switch (status.MPI_TAG) {
  case 0: MPI_Comm_free(&client);
          MPI_Close_port(port_name);
          MPI_Finalize(); return 0;
  case 1: MPI_Comm_disconnect(&client);
          again = 0;
          break;
  case 2: /* do something */
          ...
  default:
    /* Unexpected message type */
    MPI_Abort(MPI_COMM_WORLD, 1);
}

Here is the client.

#include "mpi.h"
int main( int argc, char **argv ) {
  MPI_Comm server;
  double buf[MAX_DATA];
  char port_name[MPI_MAX_PORT_NAME];
  MPI_Init(&argc, &argv);
  strcpy(port_name, argv[1]);/* assume server's name is cmd-line arg */
  MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &server);

  while (!done) {
    tag = 2; /* Action to perform */
    MPI_Send(buf, n, MPI_DOUBLE, 0, tag, server);
    /* etc */
  }
  MPI_Send(buf, 0, MPI_DOUBLE, 0, 1, server);
  MPI_Comm_disconnect(&server);
  MPI_Finalize();
  return 0;
}
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 -universe_size argument to a program that starts MPI processes.
- Automatic interaction with a batch scheduler to figure out how many processors have been allocated to an application.
- An environment variable set by the user.
- Extra information passed to MPI_COMM_SPAWN through the info argument.

An implementation must document how MPI_UNIVERSE_SIZE is set. An implementation may not support the ability to set MPI_UNIVERSE_SIZE, in which case the attribute MPI_UNIVERSE_SIZE is not set. 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 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 pro-
gram, that is, one in which MPI_COMM_WORLD has size 1.

In some implementations, MPI may not be able to function without an “MPI environ-
ment.” 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 at-
tribute 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

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.

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 pro-
gram, that is, one in which MPI_COMM_WORLD has size 1.

In some implementations, MPI may not be able to function without an “MPI environ-
ment.” 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.)
Value contains an integer that overrides the default value for MPI_APPNUM in the child.

**Rationale.** When a single application is started, it is able to figure out how many processes there are by looking at the size of MPI_COMM_World. An application consisting of multiple SPMD sub-applications has no way to find out how many sub-applications there are and to which sub-application the process belongs. While there are ways to figure it out in special cases, there is no general mechanism. MPI_APPNUM provides such a general mechanism. (End of rationale.)

### 10.5.4 Releasing Connections

Before a client and server connect, they are independent MPI applications. An error in one does not affect the other. After establishing a connection with MPI_COMM_CONNECT and MPI_COMM_ACCEPT, an error in one may affect the other. It is desirable for a client 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 MPI_COMM_WORLD) or
     b. they have previously belonged to a communicator that was freed with MPI_COMM_FREE instead of MPI_COMM_DISCONNECT or
     c. they both belong to the group of the same window or 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 MPI processes into disconnected (independent) sets (equivalence classes) of processes.

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

The following additional rules apply to MPI routines in other chapters:

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

MPI_COMM_DISCONNECT(comm)
INTO 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.)

10.5. OTHER FUNCTIONALITY

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)
This call creates an intercommunicator from the union of two MPI processes which are connected by a socket. MPI\_COMM\_JOIN 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\_COMM\_JOIN is called (see below). It is the responsibility of the application to create the socket using standard socket API calls.

MPI\_COMM\_JOIN must be called by the process at each end of the socket. It does not return until both processes have called MPI\_COMM\_JOIN. 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\_COMM\_JOIN, 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\_COMM\_JOIN is called and after MPI\_COMM\_JOIN returns. More specifically, on entry to MPI\_COMM\_JOIN, a read on the socket will not read any data that was written to the socket before the remote process called MPI\_COMM\_JOIN. On exit from MPI\_COMM\_JOIN, a read will not read any data that was written to the socket before the remote process returned from MPI\_COMM\_JOIN. It is the responsibility of the application to ensure the first condition, and the responsibility of the MPI implementation to ensure the second. In a multithreaded application, the application must ensure that one thread does not access the socket while another is calling MPI\_COMM\_JOIN, or call MPI\_COMM\_JOIN concurrently.

Advice to implementors. MPI is free to use any available communication path(s) for MPI messages in the new communicator; the socket is only used for the initial handshaking. (End of advice to implementors.)

MPI\_COMM\_JOIN uses non-MPI communication to do its work. The interaction of non-MPI communication with pending MPI communication is not defined. Therefore, the result of calling MPI\_COMM\_JOIN on two connected processes (see Section 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 \( A = B(\text{map}) \), 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: \texttt{MPI\_PUT} (remote write), \texttt{MPI\_GET} (remote read) and \texttt{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 \texttt{origin} the process that performs the call, and by \texttt{target} 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.

\[ \text{MPI\_WIN\_CREATE}(base, size, disp\_unit, info, comm, win) \]

| IN base | initial address of window (choice) |
| IN size | size of window in bytes (nonnegative integer) |
| IN disp\_unit | local unit size for displacements, in bytes (positive integer) |
| IN info | info argument (handle) |
| IN comm | communicator (handle) |
| OUT win | window object returned by the call (handle) |

\[ \text{int MPI\_Win\_create(void *base, MPI\_Aint size, int disp\_unit, MPI\_Info info, MPI\_Comm comm, MPI\_Win *win)} \]

\[ \text{MPI\_WIN\_CREATE(BASE, SIZE, DISP\_UNIT, INFO, COMM, WIN, IERROR)} \]

\[ \text{static MPI::Win MPI::Win::Create(const void* base, MPI::Aint size, int disp\_unit, const MPI::Info& info, const MPI::Intracomm& comm)} \]

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.)

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.)
void MPI::Win::Free()

Frees the window object win and returns a null handle (equal to MPI_WIN_NULL). This is a collective call executed by all processes in the group associated with win. MPI_WIN_FREE(win) can be invoked by a process only after it has completed its involvement in RMA communications on window win: i.e., the process has called MPI_WIN_FENCE, or called MPI_WIN_WAIT to match a previous call to MPI_WIN_POST or called MPI_WIN_COMPLETE to match a previous call to MPI_WIN_START or called MPI_WIN_UNLOCK to match a previous call to MPI_WIN_LOCK. When the call returns, the window memory can be freed.

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.)

11.2.2 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, calls to MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag), MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag) and MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag) will return in base a pointer to the start of the window win, and will return in size and disp_unit pointers to the size and displacement unit of the window, respectively. And similarly, in C++.

In Fortran, calls to MPI_WIN_GET_ATTR(win, MPI_WIN_BASE, base, flag, ierror), MPI_WIN_GET_ATTR(win, MPI_WIN_SIZE, size, flag, ierror) and MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror) will return in base, size and disp_unit the (integer representation of) the base address, the size and the displacement unit of the window win, respectively. (The window attribute access functions are defined in Section 6.7.3, page 227.)

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

MPI_WIN_GET_GROUP(win, group)

IN win window object (handle)

OUT group group of processes which share access to the window (handle)

int MPI_Win_get_group(MPI_Win win, MPI_Group *group)

MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)

INTEGER WIN, GROUP, IERROR
MPI::Group MPI::Win::Get_group() const

 MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to create the window, associated with win. The group is returned in group.

11.3 Communication Calls

MPI supports three RMA communication calls: MPI_PUT transfers data from the caller memory (origin) to the target memory; MPI_GET transfers data from the target memory to the caller memory; and MPI_ACCUMULATE updates locations in the target memory, e.g., by adding to these locations values sent from the caller memory. These operations are nonblocking: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and at the target, when a subsequent synchronization call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.4, page 333.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call, until the subsequent synchronization call completes.

Rationale. The rule above is more lenient than for message-passing, where we do not allow two concurrent sends, with overlapping send buffers. Here, we allow two concurrent puts with overlapping send buffers. The reasons for this relaxation are

1. Users do not like that restriction, which is not very natural (it prohibits concurrent reads).
2. Weakening the rule does not prevent efficient implementation, as far as we know.
3. Weakening the rule is important for performance of RMA: we want to associate one synchronization call with as many RMA operations as possible. If puts from overlapping buffers cannot be concurrent, then we need to needlessly add synchronization points in the code.

(End of rationale.)

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 at the target. There is one exception to this rule: namely, the same location can be updated by several concurrent accumulate calls, the outcome being as if these updates occurred 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.

Rationale. The rule above is more lenient than for message-passing, where we do not allow two concurrent sends, with overlapping send buffers. Here, we allow two concurrent puts with overlapping send buffers. The reasons for this relaxation are

1. Users do not like that restriction, which is not very natural (it prohibits concurrent reads).
2. Weakening the rule does not prevent efficient implementation, as far as we know.
3. Weakening the rule is important for performance of RMA: we want to associate one synchronization call with as many RMA operations as possible. If puts from overlapping buffers cannot be concurrent, then we need to needlessly add synchronization points in the code.

(End of rationale.)

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 at the target. There is one exception to this rule: namely, the same location can be updated by several concurrent accumulate calls, the outcome being as if these updates occurred 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.
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.

```c
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)
```

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.

### MPI_PUT

**Input Parameters:**
- `origin_addr`: initial address of origin buffer (choice)
- `origin_count`: number of entries in origin buffer (nonnegative integer)
- `origin_datatype`: datatype of each entry in origin buffer (handle)
- `target_rank`: rank of target (nonnegative integer)
- `target_disp`: displacement from start of window to target buffer (nonnegative integer)
- `target_count`: number of entries in target buffer (nonnegative integer)
- `target_datatype`: datatype of each entry in target buffer (handle)
- `win`: window object used for communication (handle)

**Output Parameters:**
- `origin_datatype`: datatype of each entry in origin buffer (handle)
- `target_datatype`: datatype of each entry in target buffer (handle)
- `IERROR`: MPI error code

**Return Value:**
- `MPI_SUCCESS` if the operation was successful
- `MPI_ERRBADADDR`, `MPI_ERRMISMATCH`, `MPI_ERRNO`, `MPI_ERRTYPE`, `MPI_ERRUNAVAIL`, `MPI_ERR undefined` otherwise
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}, \texttt{target\_rank} pair. The data are written in the target buffer at address \texttt{target\_addr} = \texttt{window\_base} + \texttt{target\_disp} \times \texttt{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}, \texttt{origin\_count}, \texttt{origin\_datatype}, \texttt{target\_rank}, \texttt{tag}, \texttt{comm}, and the target process executed a receive operation with arguments \texttt{target\_addr}, \texttt{target\_count}, \texttt{target\_datatype}, \texttt{source}, \texttt{tag}, \texttt{comm}, where \texttt{target\_addr} is the target buffer address computed as explained above, and \texttt{comm} is a communicator for the group of \texttt{win}.

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 target datatype object was defined at the target process, by the same sequence of calls used to define it at the origin process. The 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 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. \textit{(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. \textit{(End of advice to implementors.)}
11.3.2 Get

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

OUT 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 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)

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

void MPI::Win::Get(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

Similar to MPI_PUT. except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The origin_datatype may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window, 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 \( A = B(map) \), where \( A \), \( B \) and \( map \) have the same distribution, and \( map \) is a permutation. To simplify, we assume a block distribution with equal size blocks.

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

INTEGER otype(p), oindex(m), & ! used to construct origin datatypes
type(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_GETExtent(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

DO i=1,p
  total(i) = total(i-1) + count(i-1)
END DO

DO i=1,p
  count(i) = 0
END DO

! compute origin and target indices of entries.
! entry i at current process is received from location
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
! j = 1..p and k = 1..m
DO i=1,m
  j = map(i)/m+1
  k = MOD(map(i),m)+1
  count(j) = count(j)+1
  oindex(total(j)+count(j)) = i
tindex(total(j) + count(j)) = k
END DO

! create origin and target datatypes for each get operation
DO i=1,p
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, oindex(total(i)+1), 
    MPI_REAL, otype(i), ierr)
  CALL MPI_TYPE_COMMIT(otype(i), ierr)
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, tindex(total(i)+1), 
    MPI_REAL, ttype(i), ierr)
  CALL MPI_TYPE_COMMIT(ttype(i), ierr)
END DO

! this part does the assignment itself
CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,p
  CALL MPI_GET(A, 1, otype(i), i-1, 0, 1, ttype(i), win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win, ierr)
DO i=1,p
  CALL MPI_TYPE_FREE(otype(i), ierr)
  CALL MPI_TYPE_FREE(ttype(i), ierr)
END DO
RETURN
END

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)
  CALL MPI_WIN_FENCE(0, win, ierr)
  DO i=1,m
    j = map(i)/p
    k = MOD(map(i),p)
    CALL MPI_TYPE_GET_SIZE(MPI_REAL, lowerbound, sizeofreal, ierr)
    CALL MPI_WIN_CREATE(B, j*sizeofreal, sizeofreal, MPI_INFO_NULL, 
      comm, win, ierr)
    CALL MPI_WIN_FENCE(0, win, 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)

void MPI::Win::Accumulate(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 MPI::Op& op) const

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
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 \( B(j) = \sum_{i=1}^{m} A(i) \). 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)
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 \( B(j) = \sum_{i=1}^{m} A(i) \). 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)
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 Synchro-

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 synchro-

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 synchro-

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,
MPI provides three synchronization mechanisms:

1. The \texttt{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 \texttt{MPI\_WIN\_FENCE}. A process can access windows at all processes in the group of \texttt{win} during such an exposure epoch.

2. The four functions \texttt{MPI\_WIN\_START}, \texttt{MPI\_WIN\_COMPLETE}, \texttt{MPI\_WIN\_POST} and \texttt{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 RMA 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 \texttt{MPI\_WIN\_START} and is terminated by a call to \texttt{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 \texttt{MPI\_WIN\_POST} and is completed by a call to \texttt{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 \texttt{MPI\_WIN\_LOCK} and \texttt{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 \texttt{MPI\_WIN\_LOCK} and terminated by a call to \texttt{MPI\_WIN\_UNLOCK}. Only one target window can be accessed during that epoch with \texttt{win}.

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

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

MPI provides three synchronization mechanisms:

1. The \texttt{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 \texttt{MPI\_WIN\_FENCE}. A process can access windows at all processes in the group of \texttt{win} during such an access epoch, and the local window can be accessed by all processes in the group of \texttt{win} during such an exposure epoch.

2. The four functions \texttt{MPI\_WIN\_START}, \texttt{MPI\_WIN\_COMPLETE}, \texttt{MPI\_WIN\_POST} and \texttt{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 RMA 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 \texttt{MPI\_WIN\_START} and is terminated by a call to \texttt{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 \texttt{MPI\_WIN\_POST} and is completed by a call to \texttt{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 \texttt{MPI\_WIN\_LOCK} and \texttt{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 \texttt{MPI\_WIN\_LOCK} and terminated by a call to \texttt{MPI\_WIN\_UNLOCK}. Only one target window can be accessed during that epoch with \texttt{win}.

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

Figure 11.1 shows operations occurring in the natural temporal order implied by the synchronizations: the \texttt{post} occurs before the matching \texttt{start}, and \texttt{complete} occurs before the matching \texttt{wait}. However, such strong synchronization is more than needed for correct ordering of window accesses. The semantics of MPI calls allow weak synchronization, as
Figure 11.1: Active target communication. Dashed arrows represent synchronizations (ordering of events).
Local accesses exposed to RMA
Window is accesses in target memory executed put complete
memory executed put in origin
Local window accesses

start

load

store

post

put executed in target memory

Window is exposed to RMA accesses

load

store

post

put executed in origin memory

Local window accesses

start

load

store

post

put executed in target memory

Window is exposed to RMA accesses

load

store

post

Local window accesses

Figure 11.2: Active target communication, with weak synchronization. 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)

\begin{verbatim}
IN assert     program assertion (integer)
IN win        window object (handle)
\end{verbatim}

\begin{verbatim}
int MPI_Win_fence(int assert, MPI_Win win)
\end{verbatim}

The call \texttt{MPI\_WIN\_FENCE} synchronizes \texttt{RMA} calls on \texttt{win}. The call is collective on the group of \texttt{win}. All \texttt{RMA} operations on \texttt{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. \texttt{RMA} operations on \texttt{win} started by a process after the fence call returns will access their target window only after \texttt{MPI\_WIN\_FENCE} has been called by the target process.

The call completes an \texttt{RMA} access epoch if it was preceded by another fence call and the local window was the target of \texttt{RMA} accesses between these two calls. The call starts an \texttt{RMA} access epoch if it is followed by another fence call and by \texttt{RMA} communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and by \texttt{RMA} communication calls issued between these two fence calls.

A fence call usually entails a barrier synchronization: a process completes a call to \texttt{MPI\_WIN\_FENCE} only after all other processes in the group entered their matching call. However, a call to \texttt{MPI\_WIN\_FENCE} that is known not to end any epoch (in particular, a call with \texttt{assert = MPI\_MODE\_NOPRECEDE}) does not necessarily act as a barrier.

The \texttt{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 \texttt{assert = 0} is always valid.

\textit{Advice to users.} Calls to \texttt{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.)

\begin{verbatim}
void MPI::Win::Fence(int assert) const
\end{verbatim}

The call \texttt{MPI\_WIN\_FENCE} synchronizes \texttt{RMA} calls on \texttt{win}. The call is collective on the group of \texttt{win}. All \texttt{RMA} operations on \texttt{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. \texttt{RMA} operations on \texttt{win} started by a process after the fence call returns will access their target window only after \texttt{MPI\_WIN\_FENCE} has been called by the target process.

The call completes an \texttt{RMA} access epoch if it was preceded by another fence call and the local window was the target of \texttt{RMA} accesses between these two calls. The call starts an \texttt{RMA} access epoch if it is followed by another fence call and by \texttt{RMA} communication calls issued between these two fence calls. The call starts an exposure epoch if it is followed by another fence call and by \texttt{RMA} communication calls issued between these two fence calls.

A fence call usually entails a barrier synchronization: a process completes a call to \texttt{MPI\_WIN\_FENCE} only after all other processes in the group entered their matching call. However, a call to \texttt{MPI\_WIN\_FENCE} that is known not to end any epoch (in particular, a call with \texttt{assert = MPI\_MODE\_NOPRECEDE}) does not necessarily act as a barrier.

The \texttt{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 \texttt{assert = 0} is always valid.

\textit{Advice to users.} Calls to \texttt{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.)
11.4.2 General Active Target Synchronization

**MPI_Win_start(group, assert, win)**

- **IN** group: group of target processes (handle)
- **IN** assert: program assertion (integer)
- **IN** win: window object (handle)

```c
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

- **MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)**
  - INTEGER GROUP, ASSERT, WIN, IERROR

```c
void MPI::Win::Start(const MPI::Group& group, int assert) const
```

Starts an RMA access epoch for `win`. RMA calls issued on `win` during this epoch must access only windows at processes in `group`. Each process in `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 `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.

**MPI_Win_complete(win)**

- **IN** win: window object (handle)

```c
int MPI_Win_complete(MPI_Win win)
```

- **MPI_WIN_COMPLETE(WIN, IERROR)**
  - INTEGER WIN, IERROR

```c
void MPI::Win::Complete() const
```

Completes an RMA access epoch on `win` started by a call to `MPI_Win_start`. All RMA communication calls issued on `win` during this epoch will have completed at the origin when the call returns.

`MPI_Win_complete` enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin.

Consider the sequence of calls in the example below.

**Example 11.4**

```c
MPI_Win_start(group, flag, win);
MPI_Put(..., win);
MPI_Win_complete(win);
```
The call to `MPI_Win_complete` does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to `MPI_Win_start` has matched a call to `MPI_Win_post` by the target process. This still leaves much choice to implementors. The call to `MPI_Win_start` can block until the matching call to `MPI_Win_post` occurs at all target processes. One can also have implementations where the call to `MPI_Win_post` occurs at all target processes and 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.

```c
MPI_Win_post(group, assert, win)
```

- `group` group of origin processes (handle)
- `assert` program assertion (integer)
- `win` window object (handle)

```c
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

```c
MPI_Win_post(group, assert, win, IERROR)
```

```c
void MPI::Win::Post(const MPI::Group& group, int assert) 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.

```c
MPI_Win_wait(win)
```

- `win` window object (handle)

```c
int MPI_Win_wait(MPI_Win win)
```

```c
MPI_Win_wait(WIN, IERROR)
```

```c
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` 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

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_post` occurs at all target processes, 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.

```c
MPI_Win_post(group, assert, win)
```

- `group` group of origin processes (handle)
- `assert` program assertion (integer)
- `win` window object (handle)

```c
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

```c
MPI_Win_post(group, assert, win, IERROR)
```

```c
void MPI::Win::Post(const MPI::Group& group, int assert) 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.

```c
MPI_Win_wait(win)
```

- `win` window object (handle)

```c
int MPI_Win_wait(MPI_Win win)
```

```c
MPI_Win_wait(WIN, IERROR)
```

```c
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` 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
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)
IN win window object (handle)
OUT flag success flag (logical)

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
```

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

```
Figure 11.4: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.
```
group, using wincomm. No need to wait for the completion of these sends.

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 from 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 \( G = \langle V, E \rangle \), 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(outgroup, \ldots), where outgroup = \{ \( j : ij \in E \) \} and ingroup = \{ \( j : ji \in E \) \}. 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)
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

MPI_WIN_LOCK(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)

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

Completes an RMA access epoch started by a call to MPI_WIN_LOCK(...,win). 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_UNLOCK to unlock 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.)

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 provide incorrect information. Users may always provide assert = 0 to indicate a general case, where no guarantees are made.

Advice to users. Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent, shared memory machines. Users should consult their implementation 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 com-
pleted 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 oc-
curred 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 the assertion is given by any process in the window group, then it
must be given by all processes in the group.

MPI_WIN_LOCK:

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 com-
pleted 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 oc-
curred 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 the assertion is given by any process in the window group, then it
must be given by all processes in the group.
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.

```c
... 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.

```c
... 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, fromtype[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.

```c
... 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, fromtype[i], win);
}
```
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 \( A_0 \) is updated using values of array \( A_1 \), 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 \( w_{ni} \) consists of array \( A_i \).

```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 A0 (and A1) */
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
  if (!converged(A0, A1))
    MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
  MPI_Win_complete(win0); MPI_Win_complete(win1); MPI_Win_wait(win0);
  A process posts the local window associated with win0 before it completes RMA accesses to the remote windows associated with win1. When the wait(win1) 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.
11.6.2 Error Classes

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

- **MPI_ERR_WIN**: invalid win argument
- **MPI_ERR_BASE**: invalid base argument
- **MPI_ERR_SIZE**: invalid size argument
- **MPI_ERR_DISP**: invalid disp argument
- **MPI_ERR_LOCKTYPE**: invalid locktype argument
- **MPI_ERR_ASSERT**: invalid assert argument
- **MPI_ERR_RMA_CONFLICT**: conflicting accesses to window
- **MPI_ERR_RMA_SYNC**: wrong synchronization of RMA calls

Table 11.1: Error classes in one-sided communication routines

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).

### 11.7 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 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.

Table 11.1: Error classes in one-sided communication routines

### 11.7. SEMANTICS AND CORRECTNESS

The following rules specify the latest time at which an operation must complete at the origin or the target. The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier). The rules also specify the latest time at which an update of one window copy becomes visible in another overlapping copy.

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.

Table 11.1: Error classes in one-sided communication routines
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 become 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, \texttt{win1} and \texttt{win2}. A call to \texttt{MPI\_WIN\_FENCE(0, win1)} by the window owner makes visible the process memory previous updates to window \texttt{win1} 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.

\textbf{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.)

\textbf{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 until the put or accumulate update becomes visible in process memory.

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.

\textbf{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.)

\textbf{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 until the put or accumulate update becomes visible in process memory.
(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. 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.

**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 were 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

(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 occurs, 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 occurs, 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.
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. The MPI forum decided not to decide which interpretation of the standard is the correct one, since the issue is very contentious, and a decision would have much impact on implementors but less impact on users. (*End of rationale.*)

### 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
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:

```
Source of Process 1  Source of Process 2  Executed in Process 2
bbbb = 777    buff = 999
call MPI_WIN_FENCE call MPI_WIN_FENCE
reg_A:=999     call MPI_PUT(bbbb)
                into buff of process 2)
                stop appl. thread
buff:=777 in PUT handler
continue appl. thread
ccc = buff       call MPI_WIN_FENCE
ccc:=reg_A
```

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 affects 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_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

Rationale. It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is 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_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

Rationale. It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is 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)
```

<table>
<thead>
<tr>
<th>IN</th>
<th>query_fn</th>
<th>callback function invoked when request status is queried (function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>free_fn</td>
<td>callback function invoked when request is freed (function)</td>
</tr>
<tr>
<td>IN</td>
<td>cancel_fn</td>
<td>callback function invoked when request is cancelled (function)</td>
</tr>
<tr>
<td>IN</td>
<td>extra_state</td>
<td>extra state</td>
</tr>
<tr>
<td>OUT</td>
<td>request</td>
<td>generalized request (handle)</td>
</tr>
</tbody>
</table>

```c
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)
```

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);
```

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

...
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 to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer valid. However, user copies of this handle are valid until after free_fn completes since MPI does not deallocate the object until then. Since free_fn is not called until after MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this call. Users should note that MPI will deallocate the object after free_fn executes. 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 generalization request the error code returned by the corresponding invocation of its free_fn callback function. However, if the MPI function was passed MPI::STATUSES_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query_fn must not set the error field of status since query_fn may be called by MPI::WAIT or MPI::TEST, in which case the error field of status should not change. The MPI library knows the “context” in which query_fn is invoked and 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 to be set to MPI_REQUEST_NULL. This handle to the generalized request is no longer valid. However, user copies of this handle are valid until after free_fn completes since MPI does not deallocate the object until then. Since free_fn is not called until after MPI_GREQUEST_COMPLETE, the user copy of the handle can be used to make this call. Users should note that MPI will deallocate the object after free_fn executes. 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 generalization request the error code returned by the corresponding invocation of its free_fn callback function. However, if the MPI function was passed MPI::STATUSES_IGNORE, then the individual error codes returned by each callback functions will be lost.

Advice to users. query_fn must not set the error field of status since query_fn may be called by MPI::WAIT or MPI::TEST, in which case the error field of status should not change. The MPI library knows the “context” in which query_fn is invoked and
can decide correctly when to put in the error field of status the returned error code.
(End of advice to users.)

MPI_REQUEST_COMPLETE(request)
INOUT request generalized request (handle)

int MPI_Request_complete(MPI_Request request)

MPI_REQUEST_COMPLETE(REQUEST, IERROR)
INTEGER REQUEST, IERROR

void MPI::Grequest::Complete()

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_REQUEST_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_REQUEST_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.)

12.2.1 Examples

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;

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; 
    /* 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;
    /* choose not to return a value for this */
    status->MPI_SOURCE = MPI_UNDEFINED;
    /* tag has no meaning for this generalized request */
    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\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 MPI call fills in the appropriate fields in the status object. Any unused fields will have undefined values. A call to MPI_test[WAIT]{ANY|SOME|ALL} can modify any of the fields in the status object. Specifically, it can modify fields that are undefined. The
fields with meaningful 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
MPI_STATUS_SET_ELEMENTS(status, datatype, count)

INOUT status        status with which to associate count (Status)
IN datatype         datatype associated with count (handle)
IN count            number of elements to associate with status (integer)
```

```c
int MPI_Status_set_elements(MPI_Status *status, MPI_Datatype datatype, int count)
MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, 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(status, datatype, count) or to MPI_GET_ELEMENTS(status, datatype, count) must use a datatype argument that has the same type signature as the datatype argument that was used in the call to MPI_STATUS_SET_ELEMENTS.

**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
MPI_STATUS_SET_CANCELLED(status, flag)

INOUT status        status with which to associate cancel flag (Status)
IN flag             if true indicates request was cancelled (logical)
```

```c
int MPI_Status_set_cancelled(MPI_Status *status, int flag)
MPI_STATUS_SET_CANCELLED(STATUS, 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.)

```c
MPI_STATUS_SET_CANCELLED(status, flag)

INOUT status        status with which to associate cancel flag (Status)
IN flag             if true indicates request was cancelled (logical)
```

```c
int MPI_Status_set_cancelled(MPI_Status *status, int flag)
MPI_STATUS_SET_CANCELLED(STATUS, FLAG)
```
12.4. MPI AND THREADS

LOGICAL FLAG
void MPI::Status::Set_cancelled(bool flag)

If flag is set to true then a subsequent call to MPI_TEST_CANCELL ED(status, flag) will also return flag = true, otherwise it will return false.

Advice to users. Users are advised not to reuse the status fields for values other than those for which they were intended. Doing so may lead to unexpected results when using the status object. For example, calling MPI_GET_ELEMENTS may cause an error if the value is out of range or it may be impossible to detect such an error. The extra_state argument provided with a generalized request can be used to return information that does not logically belong in status. Furthermore, modifying the values in a status set internally by MPI, e.g., MPI_RECV, may lead to unpredictable results and is strongly discouraged. (End of advice to users.)

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). (End of rationale.)

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). (End of rationale.)

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(buff1, count, type, 0, 0, comm), whereas the second thread executes a blocking receive call MPI_Recv(buff2, count, type, 0, 0, comm, &status), i.e., the first thread sends a message that is received by the second thread. This communication should always succeed. According to the first requirement, the execution will correspond to some interleaving of the two calls. According to the second requirement, a call can only block the calling thread and cannot prevent progress of the other thread. If the send call went ahead of the receive call, then the sending thread may block, but this will not prevent the receiving thread from executing. Thus, the receiving 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 requests of two concurrent MPI_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test 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; 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)
```

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 lowest 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.)

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.

```c
MPI_QUERY_THREAD(provided)
OUT provided provided level of thread support (integer)
```

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

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().

```c
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)

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 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 parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. In addition, further efficiencies can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). The I/O environment described in this chapter provides these facilities.

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach in which data partitioning is expressed using derived datatypes. Compared to a limited set of predefined access patterns, this approach has the advantage of added flexibility and expressiveness.

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 displacements 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

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

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

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.
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, IERROR)

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.

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, IERROR)

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.
The format for specifying the file name in the file argument is implementation dependent and must be documented by the implementation.

Advice to implementors. An implementation may require that file 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 file 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_UNICODE_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

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
```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

`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)
```

The `MPI_FILE_DELETE` routine 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)

INOUT fh file handle (handle)
IN size size to truncate or expand file (integer)

int MPI_File_set_size(MPI_File fh, MPI_Offset size)

MPI_FILE_SET_SIZE(fh, SIZE, IERROR)

INTEGER fh, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE

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

MPI_FILE_SET_SIZE resizes the file associated with the file handle fh. size is measured in bytes from the beginning of the file. MPI_FILE_SET_SIZE is collective; all processes in the group must pass identical values for size.

If size is smaller than the current file size, the file is truncated at the position defined by size. The implementation is free to deallocate file blocks located beyond this position.

If size is larger than the current file size, the file size becomes size. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and size) are undefined. It is implementation dependent whether the MPI_FILE_SET_SIZE routine allocates file space—use MPI_FILE_PREALLOCATE to force file space to be reserved.

MPI_FILE_SET_SIZE does not affect the individual file pointers or the shared file pointer. If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, it is erroneous to call this routine.

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)

INOUT fh file handle (handle)
IN size size to preallocate file (integer)

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(FH, SIZE, IERROR)
INTEGER FH, IERROR
INTEGER(KIND=MPI_OFFSET_KIND) SIZE
MPI::Offset MPI::File::Get_size() const

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). (End of advice to users.)

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_GROUP returns, in group, the group which opened the file associated with the file handle fh. As far as consistency semantics are concerned, MPI_FILE_GET_GROUP is a data access operation (see Section 13.6.1, page 420). (End of advice to users.)
MPI::Group MPI::File::Get_group() const

MPI_FILE_GET_GROUP(FH, GROUP, IERROR)

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.

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

```fortran
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
```

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

```fortran
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
```

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.)

```
MPI_FILE_SET_INFO(fh, info)
INOUT fh file handle (handle)
IN info info object (handle)
```

```
int MPI_File_set_info(MPI_File fh, MPI_Info 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.)

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.)

```
MPI_FILE_SET_INFO(fh, info)
INOUT fh file handle (handle)
IN info info object (handle)
```

```
int MPI_File_set_info(MPI_File fh, MPI_Info 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.)
nodes cb, and size buffer parameters are further directed via additional hints:

**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 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 the system 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 access_style key 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.

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.

Advice to users. The info object returned in info_used will contain all hints currently active for this file. This set of hints may be greater or smaller than the set of hints passed in to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set. (End of advice to users.)

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 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 access_style key 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. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).

chunked_item (comma separated list of integers) [SAME]: This hint specifies the size of each array entry, in bytes.

chunked_size (comma separated list of integers) [SAME]: This hint specifies the 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_CLOSE.

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.

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**

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

This function allows the view to be changed for these types of files.

The **MPI_FILE_SET_VIEW** routine changes the process' view of the data in the file. The start of the view is set to `disp`, the type of data is set to `etype`, and the representation of data in the file is set to `datarep`. In addition, **MPI_FILE_SET_VIEW** resets the individual file pointers and the shared file pointer to zero. **MPI_FILE_SET_VIEW** is collective; the values for `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, 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` mode was specified when the file was opened, the special displacement `MPI_DISPLACEMENT_CURRENT` must be passed in `disp`. This sets the displacement to the current position of the shared file pointer. `MPI_DISPLACEMENT_CURRENT` is invalid unless the amode for the file has `MPI_MODE_SEQUENTIAL` set.

### Rationale

For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the displacement may not be meaningful.

`MPI_DISPLACEMENT_CURRENT` allows the view to be changed for these types of files.

(End of rationale.)

---

**FILE VIEWS**

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

The **MPI_FILE_SET_VIEW** routine changes the process' view of the data in the file. The start of the view is set to `disp`, the type of data is set to `etype`, and the representation of data in the file is set to `datarep`. In addition, **MPI_FILE_SET_VIEW** resets the individual file pointers and the shared file pointer to zero. **MPI_FILE_SET_VIEW** is collective; the values for `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, 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` mode was specified when the file was opened, the special displacement `MPI_DISPLACEMENT_CURRENT` must be passed in `disp`. This sets the displacement to the current position of the shared file pointer. `MPI_DISPLACEMENT_CURRENT` is invalid unless the amode for the file has `MPI_MODE_SEQUENTIAL` set.

### Rationale

For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the displacement may not be meaningful.

`MPI_DISPLACEMENT_CURRENT` allows the view to be changed for these types of files.

(End of rationale.)
Advice to implementors. It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (End of advice to implementors.)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

Advice to users. disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

```
first view: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
second view: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

file structure:

```
header: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

Figure 13.3: Displacements

(End of advice to users.)

An etype (elementary datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype. Derived etypes can be constructed by using any of the MPI datatype constructor routines, provided all resulting typemap displacements are 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.

Advice to users. In order to ensure interoperability in a heterogeneous environment, additional restrictions must be observed when constructing the etype (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.
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 (noncollective 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.

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)

<table>
<thead>
<tr>
<th>IN</th>
<th>fh</th>
<th>file handle (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>disp</td>
<td>displacement (integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>etype</td>
<td>elementary datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>filetype</td>
<td>filetype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>datarep</td>
<td>data representation (string)</td>
</tr>
</tbody>
</table>

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

MPI_FILE_GET_VIEW returns the process’s view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

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)

<table>
<thead>
<tr>
<th>IN</th>
<th>fh</th>
<th>file handle (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>disp</td>
<td>displacement (integer)</td>
</tr>
<tr>
<td>OUT</td>
<td>etype</td>
<td>elementary datatype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>filetype</td>
<td>filetype (handle)</td>
</tr>
<tr>
<td>OUT</td>
<td>datarep</td>
<td>data representation (string)</td>
</tr>
</tbody>
</table>

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

MPI_FILE_GET_VIEW returns the process’s view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.
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 or 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,

$$\text{new} \_\text{file} \_\text{offset} = \text{old} \_\text{file} \_\text{offset} + \frac{\text{elements(etype)}}{\text{elements(datatype)}} \times \text{count}$$

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 \text{datatype}, and \text{old} \_\text{file} \_\text{offset} is the value of the implicit offset before the call. The file position, \text{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 \text{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 \text{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\_ALL} or a pair of \text{MPI\_FILE\_BEGIN} and \text{MPI\_FILE\_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, \text{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: \text{buf}, \text{count}, and \text{datatype}.

Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass \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_CANCELLER 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.

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_CANCELLER 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.
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)

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)

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)

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)

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)

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)

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)

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)

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)
MPI_FILE_READ_AT_ALL is a collective version of the blocking MPI_FILE_READ_AT interface.

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

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 status status object (Status)

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

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

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf,
int count, 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_FILE_WRITE_AT interface.

MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)

IN 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, IERROR INTEGER(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_READ_AT interface.

MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)

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, IERROR INTEGER(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 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.

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

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

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

! 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.

MPI_FILE_READ_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 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.
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, IERROR)
<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)

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

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

<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

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

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

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

<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

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

MPI_FILE_WRITE_ALL writes a file using the individual file pointer.

MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE interface.

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

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

<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

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

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

MPI_FILE_WRITE_ALL writes a file using the individual file pointer.

MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE interface.
13.4. DATA ACCESS

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

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

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

! Read the first twenty real words in a file into two local! buffers. Note that when the first MPI_FILE_IREAD returns,! the file pointer has been updated to point to the! eleventh real word in the file.

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**

```
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)
```

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

**MPI_FILE_IWRITE** is a nonblocking version of the **MPI_FILE_WRITE** interface.

**MPI_FILE_SEEK**

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

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

**MPI_FILE_SEEK** updates the individual file pointer according to **whence**, which has the following possible values:

- **MPISEEK_SET**: the pointer is set to **offset**
- **MPISEEK_CUR**: the pointer is set to the current pointer position plus **offset**
- **MPISEEK_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**

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

```c
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```
calls were serialized, and

• the effect of multiple calls to shared file pointer routines is defined to behave as if the calls were serialized, and

MPI is defined to be the current value of the offset routines described in Section 13.4.2, page 390, with the following modifications:

390, page 13.4.2

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

13.4. DATA ACCESS

int MPI_File_get_position(MPI_File 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 pointer 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_FILE_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

13.4. DATA ACCESS

int MPI_File_get_position(MPI_File 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 pointer 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_FILE_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)

<type> 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)

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)

<type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

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_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_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED  
interface.

MPI_FILE_IWRITE_SHARED is a nonblocking version of the MPI_FILE_WRITE_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 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)

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

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

<integer> BUF(*)INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR

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

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

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)

<integer> OFFSET(integer) FH, 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_FILE_SEEK_SHARED is a collective version of the MPI_FILE_SEEK_SHARED interface.

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)

<integer> OFFSET(integer) FH, 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)

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

int MPI\_File\_get\_position\_shared(MPI\_File fh, MPI\_Offset offset)

MPI\_FILE\_GET\_POSITION\_SHARED\_SHARED(fh, offset, ierror)

INTEGER fh, ierror
INTEGER(KIND=MPI\_OFFSET\_KIND) offset

MPI\_Offset MPI\_File\_get\_position\_shared\_const

MPI\_FILE\_GET\_POSITION\_SHARED\_SHARED returns, in offset, the current position of the shared file pointer in byte units relative to the current view.

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.
1. Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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.

### MPI_FILE_READ_AT_ALL_BEGIN

- **fh**: file handle (handle)
- **offset**: file offset (integer)
- **buf**: initial address of buffer (choice)
- **count**: number of elements in buffer (integer)
- **datatype**: datatype of each buffer element (handle)

```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**: file handle (handle)
- **OFFSET**: file offset (integer)
- **BUF**: initial address of buffer (choice)
- **COUNT**: number of elements in buffer (integer)
- **DATATYPE**: datatype of each buffer element (handle)

```c
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT, DATATYPE, IERROR)
```

### MPI::File::Read_at_all_begin

- **offset**: file offset (integer)
- **buf**: initial address of buffer (choice)
- **count**: number of elements in buffer (integer)
- **datatype**: datatype of each buffer element (handle)

```c
void MPI::File::Read_at_all_begin(MPI::Offset offset, void* buf, int count,
                                   const MPI::Datatype& datatype)
```

### MPI_FILE_READ_AT_ALL_END

- **fh**: file handle (handle)
- **buf**: initial address of buffer (choice)
- **status**: status object (Status)

```c
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
```

### MPI_FILE_READ_AT_ALL_END

- **fh**: file handle (handle)
- **buf**: initial address of buffer (choice)
- **status**: status object (Status)

```c
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
```

### MPI::File::Read_at_all_end

- **buf**: initial address of buffer (choice)
- **status**: status object (Status)

```c
void MPI::File::Read_at_all_end(void* buf, MPI::Status& status)
void MPI::File::Read_at_all_end(void* buf)
```
13.4. DATA ACCESS

MPFILE_WRITE_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

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)

int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
int count, MPI_Datatype datatype)

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

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

MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)

<type> BUF(*)INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Write_at_all_end(const void* buf, MPI::Status& status)

MPI_FILE_READ_AT_ALL_BEGIN(fh, buf, status)

INOUT fh
file handle (handle)

IN buf
initial address of buffer (choice)

OUT status
status object (Status)

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)

<type> BUF(*)
INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::File::Read_at_all_end(const void* buf, MPI::Status& status)

void MPI::File::Read_at_all_end(const void* buf)

MPI_FILE_READ_AT_ALL_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_all_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)

MPI_FILE_READ_AT_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)

MPI_FILE_READ_AT_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
### CHAPTER 13. I/O

```c
<int type> BUF(*)
INTEGER FH, COUNT, DATATYPE, IERROR

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

MPI_FILE_READ_ALL_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_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)
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)

MPI_FILE_WRITE_ALL_BEGIN(fh, buf, count, datatype)
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)

int MPI_File_write_all_begin(MPI_File fh, void *buf, int count,
MPI_Datatype datatype)
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)

void MPI::File::Write_all_begin(const void* buf, int count,
const MPI::Datatype& datatype)
MPI_FILE_WRITE_ALL_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)

int MPI_File_write_all_end(MPI_File fh, void *buf, MPI_Status *status)
MPI_FILE_WRITE_ALL_END(fh, buf, status)
INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
OUT status status object (Status)
```
13.4. DATA ACCESS

MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
   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, const MPI::Datatype& datatype)

MPI_FILE_READ_ORDERED_BEGIN(FH, BUF, COUNT, DATATYPE, IERROR)
   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)
   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)
  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)

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)
  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)
  INOUT 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.)

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.)

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.)
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.

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.
file. For etypes and filetypes 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”, “external32”, 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)

IN   fh
file handle (handle)
IN   datatype
datatype (handle)
OUT  extent
datatype extent (integer)
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

Returns the extent of datatype in the file fh. This extent will be the same for all processes accessing the file fh. If the current view uses a user-defined data representation (see Section 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 in big-endian IEEE format [27] of the appropriate size. 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.)
13.5. FILE INTEROPERABILITY

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.

```
MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn, dtype_file_extent_fn, extra_state)
```

- **IN** `datarep` data representation identifier (string)
- **IN** `read_conversion_fn` function invoked to convert from file representation to native representation (function)
- **IN** `write_conversion_fn` function invoked to convert from native representation to file representation (function)
- **IN** `dtype_file_extent_fn` function invoked to get the extent of a datatype as represented in the file (function)
- **IN** `extra_state` extra state

```
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)
```

```
void MPI::Register_datarep(const char* datarep, 
    MPI::Datarep_conversion_function* read_conversion_fn, 
    MPI::Datarep_conversion_function* write_conversion_fn, 
    MPI::Datarep_conversion_function* dtype_file_extent_fn, 
    void* 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.

```
MPI_REGISTER_DATAREP(datarep, read_conversion_fn, write_conversion_fn, 
    dtype_file_extent_fn, extra_state)
```

- **IN** `datarep` data representation identifier (string)
- **IN** `read_conversion_fn` function invoked to convert from file representation to native representation (function)
- **IN** `write_conversion_fn` function invoked to convert from native representation to file representation (function)
- **IN** `dtype_file_extent_fn` function invoked to get the extent of a datatype as represented in the file (function)
- **IN** `extra_state` extra state

```
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)
```

```
void MPI::Register_datarep(const char* datarep, 
    MPI::Datarep_conversion_function* read_conversion_fn, 
    MPI::Datarep_conversion_function* write_conversion_fn, 
    MPI::Datarep_conversion_function* dtype_file_extent_fn, 
    void* extra_state)
```
<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</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_CHARACTER</td>
<td>1</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>4</td>
</tr>
<tr>
<td>MPI_INTEGER</td>
<td>4</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_REAL4</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
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

```c
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
                                       MPI_Aint *file_extent,
                                       void *extra_state);

SUBROUTINE DATAREPExtent_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
  INTEGER DATATYPE, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
  void *extra_state
end subroutine DATAREPExtent_FUNCTION

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

```c
typedef int MPI_Datarep_conversion_function(void *userbuf,
                                             MPI_Datatype datatype,
                                             int count,
                                             void *filebuf,
                                             MPI_Offset position,
                                             void *extra_state);

SUBROUTINE DATAREPConversion_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
  void *extra_state
end subroutine DATAREPConversion_FUNCTION

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...
contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in $extra_state$, the argument that was passed to the MPI_REGISTER_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

**Advise 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.)

**Advise 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 conversion 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.)

**Advise 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.)

count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in $extra_state$, the argument that was passed to the MPI_REGISTER_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

**Advise 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.)

**Advise 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 conversion 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.)

**Advise 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 else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or write routines in Section 13.4, page 387, or MPI_FILE_GET_TYPE_EXTENT is called 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 else by making repeated calls to the conversion function with the same datatype argument and appropriate values for position.

An implementation will only invoke the callback routines in this section (read_conversion_fn, write_conversion_fn, and dtype_file_extent_fn) when one of the read or write routines in Section 13.4, page 387, or MPI_FILE_GET_TYPE_EXTENT is called 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 \((e.g., fh_{1a} \in FH_1 \text{ and } fh_{1b} \in FH_1)\), and two file handles from different collective opens \((e.g., fh_1 \in FH_1 \text{ and } fh_2 \in FH_2)\).

For the purpose of consistency semantics, a matched pair (Section 13.4.5, page 404) of split collective data access operations \((e.g., \text{MPI\_FILE\_READ\_ALL\_BEGIN} \text{ and } \text{MPI\_FILE\_READ\_ALL\_END})\) compose a single data access operation. Similarly, a non-blocking data access routine \((e.g., \text{MPI\_FILE\_IREAD})\) and the routine which completes the request \((e.g., \text{MPI\_WAIT})\) 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 of split collective data access operations \((e.g., \text{MPI\_FILE\_READ\_ALL\_BEGIN} \text{ and } \text{MPI\_FILE\_READ\_ALL\_END})\) compose a single data access operation. Similarly, a non-blocking data access routine \((e.g., \text{MPI\_FILE\_IREAD})\) and the routine which completes the request \((e.g., \text{MPI\_WAIT})\) 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.
13.6. CONSISTENCY AND SEMANTICS  421

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 $SEQ_{1,2}$ 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 $MPI_FILE_SYNC()$.) $SEQ_{1,2}$ 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: $fh_{1,2} \in FH_1$ and $fh_{1,2} \in FH_2$. Assume $A_1$ is a data access operation using $fh_{1,2}$, and $A_2$ is a data access operation using $fh_{1,2}$. 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 $SEQ_{1,2}$ to the file, there is no sequence $SEQ_{2,1}$ to the file which is concurrent with $SEQ_{1,2}$. 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.

MPLFILE_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)

MPLFILE_SET_ATOMICITY(FH, FLAG, IERROR)

INTEGER FH, IERROR

LOGICAL FLAG

overlab 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 $SEQ_{1,2}$ 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 $MPI_FILE_SYNC()$.) $SEQ_{1,2}$ 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: $fh_{1,2} \in FH_1$ and $fh_{1,2} \in FH_2$. Assume $A_1$ is a data access operation using $fh_{1,2}$, and $A_2$ is a data access operation using $fh_{1,2}$. 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 $SEQ_{1,2}$ to the file, there is no sequence $SEQ_{2,1}$ to the file which is concurrent with $SEQ_{1,2}$. 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 $F_H$ be the set of file handles created by one collective open. The consistency semantics for data access operations using $F_H$ is set by collectively calling `MPI_FILE_SET_ATOMICITY` on $F_H$. `MPI_FILE_SET_ATOMICITY` is collective; all processes in the group must pass identical values for $fh$ and $flag$. If $flag$ is true, atomic mode is set; if $flag$ 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 `MPI_WAIT`) are only guaranteed to abide by nonatomic mode consistency semantics.

**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.)

```
void MPI::File::Set_atomicity(bool flag)

Let $F_H$ be the set of file handles created by one collective open. The consistency semantics for data access operations using $F_H$ is set by collectively calling `MPI_FILE_SET_ATOMICITY` on $F_H$. `MPI_FILE_SET_ATOMICITY` is collective; all processes in the group must pass identical values for $fh$ and $flag$. If $flag$ is true, atomic mode is set; if $flag$ 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 `MPI_WAIT`) are only guaranteed to abide by nonatomic mode consistency semantics.

**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.)

```

```
MPI_FILE_GET_ATOMICITY(fh, flag)

IN fh file handle (handle)
OUT flag true if atomic mode, false if nonatomic mode (logical)

int MPI_File_get_atomicity(MPI_File fh, int *flag)

```

```
MPI_FILE_GET_ATOMICITY(fh, flag)

IN fh file handle (handle)
OUT flag true if atomic mode, false if nonatomic mode (logical)

int MPI_File_get_atomicity(MPI_File fh, int *flag)

```

```
MPI_FILE_SYNC(fh)

INOUT fh file handle (handle)

int MPI_File_sync(MPI_File fh)

```

```
MPI_FILE_SYNC(fh)

INOUT fh file handle (handle)

int MPI_File_sync(MPI_File fh)

```

Calling `MPI_FILE_SYNC` with $fh$ causes all previous writes to $fh$ by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of $fh$ by the calling process.

Calling `MPI_FILE_SYNC` with $fh$ causes all previous writes to $fh$ by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of $fh$ by the calling process.
13.6. CONSISTENCY AND SEMANTICS

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 implementation. 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).
13.6.9 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 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.

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_File_write_at(fh0, 0, a, 10, MPI_INT, &status); /* MPI_Barrier( MPI_COMM_WORLD ) */

/* Process 1 */
int b[10];
int TRUE = 1;
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_read_at(fh1, 0, b, 10, MPI_INT, &status);

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:

/* 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 );
MPI_File_sync( fh0 );

/* Process 1 */
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 );
MPI_File_sync( fh0 );

/* Process 1 */
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 );
MPI_File_sync( fh0 );
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 eliminating the apparently superfluous second “sync” call for each process.

```c
/* ---------------- 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 temporally 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_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
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 serial consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then serial 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_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_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[1]) ;
MPI_Wait(&reqs[0], &status) ;
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_iread_at(fh, 10, &b, 1, MPI_INT, &reqs[0]) ;
MPI_Wait(&reqs[0], &status) ;
MPI_Wait(&reqs[1], &status) ;
```
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
1. 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 first 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 first 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 first 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 first 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.)
13.9. EXAMPLES

Table 13.3: I/O Error Classes

<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
### CHAPTER 13. I/O

* Parameters:*
  - fh previously opened MPI file handle
  - buftype MPI datatype for memory layout
  - (Assumes a compatible view has been set on fh)
  - bufcount # buftype elements to transfer

---

```c
/* this macro switches which buffer "x" is pointing to */
#define TOGGLE_PTR(x) (((x) ==(buffer1)) ? (x=buffer2) : (x=buffer1))

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_begin(fh, write_buf_ptr, bufcount, buftype);
        TOGGLE_PTR(write_buf_ptr);
        MPI_File_write_all_end(fh, write_buf_ptr, &status);
        TOGGLE_PTR(compute_buf_ptr);
        MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
    }
}
```

---

```c
/* this macro switches which buffer "x" is pointing to */
#define TOGGLE_PTR(x) (((x) ==(buffer1)) ? (x=buffer2) : (x=buffer1))

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, &status);
        TOGGLE_PTR(write_buf_ptr);
        MPI_File_write_all_begin(fh, write_buf_ptr, bufcount, buftype);
    }
}
```
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);
```

13.9.2 Subarray Filetype Constructor

![Figure 13.4: Example array file layout](image)

![Figure 13.5: Example local array filetype for process 1](image)
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, MPIDOUBLE_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.

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 MPI_ 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 imple-
   menting 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.
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

```
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;
    ...
```

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

```
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...
The same source file can then be compiled to produce both versions of the library, depending on the state of the PROFILELIB macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is 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

```bash
% cc ... -lmyprof -lpmpi -lmpi
```

Here `libmyprof.a` contains the profiler functions that intercept some of the MPI functions, `libpmpi.a` contains the "name shifted" MPI functions, and `libmpi.a` contains the normal definitions of the MPI functions.

### 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 profiling 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

```c
int FUNCTION(MPI_Example)(/* appropriate args */){
  /* Useful content */
}
```
of the profiled entry points will be undefined when the profiling library is called. Therefore none of the profiling code will be included in the image. When the standard MPI library is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of the MPI functions. The overall effect is that the code will link successfully, but will not be profiled.

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be moved out of the base library and into the profiling one.

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, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

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.

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, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

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(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

The following function is deprecated and is superseded by

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)
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.

\begin{verbatim}
MPI\_ADDRESS(location, address)
IN   location   location in caller memory (choice)
OUT  address   address of location (integer)

int MPI\_Address(void* location, MPI\_Aint *address)

MPI\_ADDRESS(LOCATION, ADDRESS, IERROR)
<type> LOCATION(*)INTEGER ADDRESS, IERROR
\end{verbatim}

The following functions are deprecated and are superseded by MPI\_TYPE\_GET\_EXTENT in MPI-2.0.

\begin{verbatim}
MPI\_TYPE\_EXTENT(datatype, extent)
IN   datatype   datatype (handle)
OUT  extent    datatype extent (integer)

int MPI\_Type\_extent(MPI\_Datatype datatype, MPI\_Aint *extent)

MPI\_TYPE\_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR
\end{verbatim}

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.

\begin{verbatim}
MPI\_TYPE\_LB( datatype, displacement)
IN   datatype   datatype (handle)
OUT  displacement   displacement of lower bound from origin, in bytes (integer)

int MPI\_Type\_lb(MPI\_Datatype datatype, MPI\_Aint* displacement)

MPI\_TYPE\_LB( DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR
\end{verbatim}

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.

\begin{verbatim}
MPI\_ADDRESS(location, address)
IN   location   location in caller memory (choice)
OUT  address   address of location (integer)

int MPI\_Address(void* location, MPI\_Aint *address)

MPI\_ADDRESS(LOCATION, ADDRESS, IERROR)
<type> LOCATION(*)INTEGER ADDRESS, IERROR
\end{verbatim}

The following functions are deprecated and are superseded by MPI\_TYPE\_GET\_EXTENT in MPI-2.0.

\begin{verbatim}
MPI\_TYPE\_EXTENT(datatype, extent)
IN   datatype   datatype (handle)
OUT  extent    datatype extent (integer)

int MPI\_Type\_extent(MPI\_Datatype datatype, MPI\_Aint *extent)

MPI\_TYPE\_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR
\end{verbatim}

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.

\begin{verbatim}
MPI\_TYPE\_LB( datatype, displacement)
IN   datatype   datatype (handle)
OUT  displacement   displacement of lower bound from origin, in bytes (integer)

int MPI\_Type\_lb(MPI\_Datatype datatype, MPI\_Aint* displacement)

MPI\_TYPE\_LB( DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR
\end{verbatim}
CHAPTER 15. DEPRECATED FUNCTIONS

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)

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)

The copy_fn function is invoked when a communicator is duplicated by
MPI_COMM_CREATE. copy_fn should be of type MPI_Copy_function, which is defined as follows:

typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                              void *extra_state, void *attribute_val_in,
                              void *attribute_val_out, int *flag)

A Fortran declaration for such a function is as follows:

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.

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)

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)

The copy_fn function is invoked when a communicator is duplicated by
MPI_COMM_CREATE. copy_fn should be of type MPI_Copy_function, which is defined as follows:

typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                              void *extra_state, void *attribute_val_in,
                              void *attribute_val_out, int *flag)

A Fortran declaration for such a function is as follows:

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.
The following function is deprecated and is superseded by `MPI_ATTR_DELETE` 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.

```plaintext
MPI_ATTR_PUT(comm, keyval, attribute_val)
INOUT comm communicator to which attribute will be attached (handle)
IN keyval key value, as returned by `MPI_KEYVAL_CREATE` (integer)
IN attribute_val attribute value
```

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.

```plaintext
MPI_ATTR_GET(comm, keyval, attribute_value)
INOUT comm communicator to which attribute will be attached (handle)
IN keyval key value, as returned by `MPI_KEYVAL_CREATE` (integer)
IN attribute_val attribute value
```
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.

```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.

```c
int MPI_Errhandler_create(MPI_Handler_function *function,
                         MPI_Errhandler *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)
```

```fortran
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)  
MPI_ERRHANDLER_SET(comm, errhandler, IERROR)  
INTEGER COMM, ERRHANDLER, IERROR
```

Associates the new error handler `errhandler` with communicator `comm` at the calling process. Note that an error handler is always associated with the communicator.

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)
```

```fortran
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)  
MPI_ERRHANDLER_GET(comm, errhandler, IERROR)  
INTEGER COMM, ERRHANDLER, IERROR
```

Returns in `errhandler` (a handle to) the error handler that is currently associated with communicator `comm`. 

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 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 Intracomm : public Comm {...};
    class Graphcomm : public Intracomm {...};
    class Cartcomm : public Intracomm {...};
    class Datatype {...};
    class Errhandler {...};
    class Exception {...};
    class File {...};
    class Group {...};
    class Info {...};
    class Op {...};
    class Request : public Request {...};
    class Grequest : 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. (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
    }
};
```

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_BCST is implemented by repeated calls to MPI_SEND or MPI_RECV, the behavior of MPI_BCST cannot be changed by derived communicator classes that might redefine MPI_SEND or MPI_RECV. The implementation of MPI_BCST must explicitly use the MPI_SEND (or MPI_RECV) of the base MPI::Comm class. (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 `MPI::<CLASS>` rather than listing each function for every MPI class; the word `<CLASS>` can be replaced with any valid MPI class name (e.g., `Group`), except as noted.

Construction / Destruction The default constructor and destructor are prototyped as follows:

```cpp
MPI::<CLASS>()
~MPI::<CLASS>()
```

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 `true`. The default constructors do not create new MPI opaque objects. Some classes have a member function `Create()` for this purpose.

**Example 16.2** In the following code fragment, the test will return `true` and the message will be sent to `cout`.

```cpp
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
    }
};
```

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_BCST is implemented by repeated calls to MPI_SEND or MPI_RECV, the behavior of MPI_BCST cannot be changed by derived communicator classes that might redefine MPI_SEND or MPI_RECV. The implementation of MPI_BCST must explicitly use the MPI_SEND (or MPI_RECV) of the base MPI::Comm class. (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 `MPI::<CLASS>` rather than listing each function for every MPI class; the word `<CLASS>` can be replaced with any valid MPI class name (e.g., `Group`), except as noted.

Construction / Destruction The default constructor and destructor are prototyped as follows:

```cpp
MPI::<CLASS>()
~MPI::<CLASS>()
```

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 `true`. The default constructors do not create new MPI opaque objects. Some classes have a member function `Create()` for this purpose.

**Example 16.2** In the following code fragment, the test will return `true` and the message will be sent to `cout`. 

```cpp
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
    }
};
```
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.

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:
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.

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:

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_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

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.

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:

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_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::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::INT</td>
<td>signed int</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI::LONG_LONG</td>
<td>signed long</td>
<td>signed long</td>
</tr>
<tr>
<td>MPI::LONG_LONG</td>
<td>signed long</td>
<td>signed long</td>
</tr>
<tr>
<td>MPI::LONG_LONG</td>
<td>signed long</td>
<td>signed long</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::UNSIGNED_LONG</td>
<td>unsigned long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>MPI::FLOAT</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>MPI::DOUBLE</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>MPI::LONG_DOUBLE</td>
<td>long double</td>
<td>long double</td>
</tr>
<tr>
<td>MPI::BOOL</td>
<td>bool</td>
<td>Complex&lt;float&gt;</td>
</tr>
<tr>
<td>MPI::COMPLEX</td>
<td>Complex&lt;double&gt;</td>
<td>Complex&lt;long double&gt;</td>
</tr>
<tr>
<td>MPI::WCHAR</td>
<td>wchar_t</td>
<td>wchar_t</td>
</tr>
<tr>
<td>MPI::BYTE</td>
<td></td>
<td>MPI::PACKED</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::F_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::F_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>
16.1. C++

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::FLOAT_INT</td>
<td>C/C++ reduction type</td>
</tr>
<tr>
<td>MPI::DOUBLE_INT</td>
<td>C/C++ reduction type</td>
</tr>
<tr>
<td>MPI::LONG_INT</td>
<td>C/C++ reduction type</td>
</tr>
<tr>
<td>MPI::SHORT_INT</td>
<td>C/C++ reduction type</td>
</tr>
<tr>
<td>MPI::LONG_DOUBLE_INT</td>
<td>C/C++ reduction type</td>
</tr>
<tr>
<td>MPI::TWOREAL</td>
<td>Fortran reduction type</td>
</tr>
<tr>
<td>MPI::TWOINTEGER</td>
<td>Fortran reduction type</td>
</tr>
<tr>
<td>MPI::F_COMPLEX</td>
<td>Optional Fortran type</td>
</tr>
<tr>
<td>MPI::INTEGER1</td>
<td>Explicit size type</td>
</tr>
<tr>
<td>MPI::INTEGER2</td>
<td>Explicit size type</td>
</tr>
<tr>
<td>MPI::INTEGER4</td>
<td>Explicit size type</td>
</tr>
<tr>
<td>MPI::INTEGER8</td>
<td>Explicit size type</td>
</tr>
<tr>
<td>MPI::REAL4</td>
<td>Explicit size type</td>
</tr>
<tr>
<td>MPI::REAL8</td>
<td>Explicit size type</td>
</tr>
<tr>
<td>MPI::REAL16</td>
<td>Explicit size type</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::Intracomm.

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 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:

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 { ... }; // more definitions
        virtual Intracomm& Clone() const { ... }; // more definitions
    };
    class Intercomm : public Comm {
        Intercomm Dup() const { ... }; // more definitions
        virtual Intercomm& Clone() const { ... }; // more definitions
    };
    // 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

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>`, `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
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:

```
Example 16.6 Sample mpi.h file.

namespace PMPI {
  class Comm {
    public:
      int Get_size() const;
  };// etc.
};

namespace MPI {
  public:
    class Comm {
      public:
        int Get_size() const;
        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:

```
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:

```
Example 16.6 Sample mpi.h file.

namespace PMPI {
  class Comm {
    public:
      int Get_size() const;
  };// etc.
};

namespace MPI {
  public:
    class Comm {
      public:
        int Get_size() const;
        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:

```
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

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.

```cpp
const MPI::Intracomm MPI::COMM_WORLD;
```

Example 16.9 mpi_no_profile.cc, to be compiled into libmpi.a.

```cpp
int MPI::Comm::Get_size() const
{
    return pmpi_comm.Get_size();
}
```

Example 16.10 mpi_profile.cc, to be compiled into libpmpi.a.

```cpp
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.)

Example 16.8 constants.cc, to be compiled into libmpi.a.

```cpp
const MPI::Intracomm MPI::COMM_WORLD;
```

Example 16.9 mpi_no_profile.cc, to be compiled into libmpi.a.

```cpp
int MPI::Comm::Get_size() const
{
    return pmpi_comm.Get_size();
}
```

Example 16.10 mpi_profile.cc, to be compiled into libpmpi.a.

```cpp
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.)

Example 16.8 constants.cc, to be compiled into libmpi.a.

```cpp
const MPI::Intracomm MPI::COMM_WORLD;
```

Example 16.9 mpi_no_profile.cc, to be compiled into libmpi.a.

```cpp
int MPI::Comm::Get_size() const
{
    return pmpi_comm.Get_size();
}
```

Example 16.10 mpi_profile.cc, to be compiled into libpmpi.a.

```cpp
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 entry 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_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_ARGV_NULL, and MPI_ARGVS_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 entry 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_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_ARGV_NULL, and MPI_ARGVS_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.
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:N:2) involves only the elements of A with indices 1, 3, 5, ... . The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., B(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.¹

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:

```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.)

¹ Technically, the Fortran standards are worded to allow non-contiguous storage of any array data.
Since the first dummy argument to MPI_RECV is an assumed-size array (<type> buf(*)),
the array section a(1:100:2) is copied to a temporary before being passed to MPI_RECV,
so that it is contiguous in memory. MPI_RECV returns immediately, and data is copied
from the temporary back into the array a. Sometime later, MPI may write to the address of
the deallocated temporary. Copying is also a problem for MPI_SEND 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

real a(100)
call MPI_RECV(a(1:100:2), MPI_REAL, 50, ... )

If a is copied, MPI_RECV 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.

2To 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_RECV(a(1:100:2), MPI_REAL, 50, ... )

Since the first dummy argument to MPI_RECV is an assumed-size array (<type> buf(*)），
the array section a(1:100:2) is copied to a temporary before being passed to MPI_RECV,
so that it is contiguous in memory. MPI_RECV returns immediately, and data is copied
from the temporary back into the array a. Sometime later, MPI may write to the address of
the deallocated temporary. Copying is also a problem for MPI_SEND 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

real a(100)
call MPI_RECV(a(1:100:2), MPI_REAL, 50, ... )

If a is copied, MPI_RECV 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.

2To 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.

### 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
```
A Problem with Register Optimization

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_RECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. This section discusses register optimization pitfalls.

When a variable is local to a Fortran subroutine (i.e., not in a module or COMMON block), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register which held a valid copy of such a variable before the call will still hold a valid copy on return. Normally users are not afflicted with this. But the user should pay attention to this section if in his/her program a buffer argument to an MPI_SEND, MPI_RECV etc., uses a name which hides the actual variables involved. MPI_BOTTOM with an MPI_Datatype containing absolute addresses is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one mentioned in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.
Example 16.11 shows what Fortran compilers are allowed to do.

**Example 16.11 Fortran 90 register optimization.**

This source can be compiled as:

```fortran
call MPI_GET_ADDRESS(buf,bufaddr,...)
call MPI_TYPE_CREATE_STRUCT(1,1,bufaddr,...)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf
val_new = buf
```

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

```fortran
call MPI_RECV(MPI_BOTTOM,...)
call MPI_WAIT(req,...)
bl = buf
```

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.

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

```fortran
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

Example 16.12 shows what Fortran compilers are allowed to do.

**Example 16.12 Fortran 90 register optimization.**

This source can be compiled as:

```fortran
call MPI_GET_ADDRESS(buf,bufaddr,...)
call MPI_TYPE_CREATE_STRUCT(1,1,bufaddr,...)
call MPI_TYPE_COMMIT(type,ierror)
val_old = buf
val_new = buf
```

The compiler may be prevented from moving a reference to a buffer across a call to `MPI_RECV` 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

```fortran
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```
with the separately compiled

```fortran
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

```fortran
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 (e.g., `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.)*

```fortran
with the separately compiled

```fortran
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

```fortran
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 (e.g., `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 KBD 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.
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, CHARACTER) with an optional integer KIND parameter that selects from 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:

\[
\text{double precision } x \\
\text{real(KIND(0.0d0)) } \times
\]

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.

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, 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:

\[
\text{double precision } x \\
\text{real(KIND(0.0d0)) } \times
\]
**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 represen-
tation 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 COM-
PLEX 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 el-
lement D(p, r). Either p or r may be omitted from calls to *selected_real_kind(p, r)*.

**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 represen-
tation 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 COM-
PLEX 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 el-
lement 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 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{verbatim}
MPI\_TYPE\_CREATE\_F90\_COMPLEX(p, r, newtype)
IN p precision, in decimal digits (integer)
IN r decimal exponent range (integer)
OUT newtype the requested MPI datatype (handle)
\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 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{verbatim}
MPI\_TYPE\_CREATE\_F90\_INTEGER(r, newtype)
IN r decimal exponent range, i.e., number of decimal digits (integer)
OUT newtype the requested MPI datatype (handle)
\end{verbatim}

This function returns a predefined MPI datatype that matches an 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.

\begin{verbatim}
MPI\_TYPE\_CREATE\_F90\_COMPLEX(p, r, newtype)
IN p precision, in decimal digits (integer)
IN r decimal exponent range (integer)
OUT newtype the requested MPI datatype (handle)
\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 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:

\begin{verbatim}
MPI\_TYPE\_CREATE\_F90\_INTEGER(r, newtype)
IN r decimal exponent range, i.e., number of decimal digits (integer)
OUT newtype the requested MPI datatype (handle)
\end{verbatim}

This function returns a predefined MPI datatype that matches an 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:
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_TYPECREATE|_F90_REAL/COMPLEX/INTEGER are given by the following rules.

For MPI_TYPECREATE|_F90_REAL:

\[
\begin{align*}
\text{if} & \quad (p > 33) \text{ or } (r > 4931) \text{ then external32 representation is undefined} \\
\text{else if} & \quad (p > 15) \text{ or } (r > 307) \text{ then external32_size = 16} \\
\text{else if} & \quad (p > 6) \text{ or } (r > 37) \text{ then external32_size = 8} \\
\text{else} & \quad \text{external32_size = 4} \\
\end{align*}
\]

For MPI_TYPECREATE|_F90_COMPLEX; twice the size as for MPI_TYPECREATE|_F90_REAL.

For MPI_TYPECREATE|_F90_INTEGER:

\[
\begin{align*}
\text{if} & \quad (r > 38) \text{ then external32 representation is undefined} \\
\text{else if} & \quad (r > 18) \text{ then external32_size = 16} \\
\text{else if} & \quad (r > 4) \text{ then external32_size = 8} \\
\text{else if} & \quad (r > 2) \text{ then external32_size = 4} \\
\text{else} & \quad \text{external32_size = 1} \\
\end{align*}
\]

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_PACKEXTERNAL, MPI_UNPACKEXTERNAL 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:

\[
\begin{align*}
\text{MPI_REAL4} \\
\end{align*}
\]
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> X INTEGER 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> X INTEGER 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 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, com-
munication 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 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, com-
munication 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)
16.2. FORTRAN SUPPORT

call MPI_SIZEDOF(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, ...)
else 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 homogeneous 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 attention to the representation sizes. Consider the following code:

real(selected_real_kind(5)) x(100)
call MPI_SIZEDOF(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
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 Language Interoperability

#### 16.3.1 Introduction

It is not uncommon for library developers to use one language to develop an applications 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 Language Interoperability

#### 16.3.1 Introduction

It is not uncommon for library developers to use one language to develop an applications 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_.
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) argc, argv arguments of the C/C++ version of MPI_INIT in order to propagate values for argc and argv to all executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (End of advice to users.)

The function MPI_INITIALIZED returns the same answer in all languages.

The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

The function MPI_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

Advice to users. The use of several languages in one MPI program may require the use of special options at compile and/or link time. (End of advice to users.)

Advice to implementors. Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code 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 MPI_Fint is provided in C/C++ for an integer of the size that matches a Fortran INTEGER; usually, MPI_Fint will be equivalent to 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.

MPI_Comp MPI_Comm_f2c(MPI_Fint comm)

If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a valid C handle to that same communicator; if comm = MPI_COMM_NULL (Fortran value), then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then MPI_Comm_f2c returns an invalid C handle.

MPI_Fint MPI_Comm_c2f(MPI_Ccomp comm)

The function 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.

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) argc, argv arguments of the C/C++ version of MPI_INIT in order to propagate values for argc and argv to all executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (End of advice to users.)

The function MPI_INITIALIZED returns the same answer in all languages.

The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

The function MPI_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

Advice to users. The use of several languages in one MPI program may require the use of special options at compile and/or link time. (End of advice to users.)

Advice to implementors. Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code 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 MPI_Fint is provided in C/C++ for an integer of the size that matches a Fortran INTEGER; usually, MPI_Fint will be equivalent to 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.

MPI_Comp MPI_Comm_f2c(MPI_Fint comm)

If comm is a valid Fortran handle to a communicator, then MPI_Comm_f2c returns a valid C handle to that same communicator; if comm = MPI_COMM_NULL (Fortran value), then MPI_Comm_f2c returns a null C handle; if comm is an invalid Fortran handle, then MPI_Comm_f2c returns an invalid C handle.

MPI_Fint MPI_Comm_c2f(MPI_Ccomp comm)

The function 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.

```c
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_Request MPI_Request_c2f(MPI_Request request)
MPI_File MPI_File_f2c(MPI_Fint file)
MPI_File MPI_File_c2f(MPI_File file)
MPI_Win MPI_Win_f2c(MPI_Fint win)
MPI_Win MPI_Win_c2f(MPI_Win win)
MPI_Op MPI_Op_f2c(MPI_Fint op)
MPI_Op MPI_Op_c2f(MPI_Op op)
MPI_Info MPI_Info_f2c(MPI_Fint info)
MPI_Info MPI_Info_c2f(MPI_Info info)
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
MPI_Errhandler 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 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);
}
```

Similar functions are provided for the other types of opaque objects.

```c
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_Request MPI_Request_c2f(MPI_Request request)
MPI_File MPI_File_f2c(MPI_Fint file)
MPI_File MPI_File_c2f(MPI_File file)
MPI_Win MPI_Win_f2c(MPI_Fint win)
MPI_Win MPI_Win_c2f(MPI_Win win)
MPI_Op MPI_Op_f2c(MPI_Fint op)
MPI_Op MPI_Op_c2f(MPI_Op op)
MPI_Info MPI_Info_f2c(MPI_Fint info)
MPI_Info MPI_Info_c2f(MPI_Info info)
MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)
MPI_Errhandler 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 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);
}
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 inline 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.

```c
MPI::<CLASS>::<CLASS>(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.

```c
MPI::<CLASS>::<CLASS>::operator=(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
// 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) {
    ...
}
```

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.

```c
MPI::<CLASS>::<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.

```c
MPI::<CLASS>::<CLASS>::operator=(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
// 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) {
    ...
}
```
1 // the MPI_Comm (c_comm) is automatically promoted to MPI::Comm
2 cpp_lib_call(c_comm);
3 }
4
5 The following function allows conversion from C++ objects to C MPI handles. In this
6 case, the casting operator is overloaded to provide the functionality.
7 MPI::<CLASS>::operator MPI_<CLASS>() const
8
9 Example 16.15 A C library routine is called from a C++ program. The C library routine
10 is prototyped to take an MPI_Comm as an argument.
11 // C function prototype
12 extern "C" {
13 void c_lib_call(MPI_Comm c_comm);
14 }
15
16 void cpp_function()
17 {
18     // Create a C++ communicator, and initialize it with a dup of
19     // MPI::COMM_WORLD
20     MPI::Intracomm cpp_comm(MPI::COMM_WORLD.Dup());
21     c_lib_call(cpp_comm);
22 }
23
24 Rationale. Providing conversion from C to C++ via constructors and from C++
25 to C via casting allows the compiler to make automatic conversions. Calling C from
26 C++ becomes trivial, as does the provision of a C or Fortran interface to a C++
27 library. (End of rationale.)
28
29 Advice to users. Note that the casting and promotion operators return new handles
30 by value. Using these new handles as INOUT parameters will affect the internal MPI
31 object, but will not affect the original handle from which it was cast. (End of advice
32 to users.)
33
34 It is important to note that all C++ objects and their corresponding C handles can be
35 used interchangeably by an application. For example, an application can cache an attribute
36 on MPI_COMM_WORLD and later retrieve it from MPI::COMM_WORLD.
37
38.5 Status
39
40 The following two procedures are provided in C to convert from a Fortran status (which
41 is an array of integers) to a C status (which is a structure), and vice versa. The conversion
42 occurs on all the information in status, including that which is hidden. That is, no status
43 information is lost in the conversion.
44
45 int MPI_Status_f2c(MPI_Fint *f_status, MPI_Status *c_status)
46
47 If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE
48 or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with
49
the same content. If \texttt{f\_status} is the Fortran value of \texttt{MPI\_STATUS\_IGNORE} or \texttt{MPI\_STATUSES\_IGNORE}, or if \texttt{f\_status} is not a valid Fortran status, then the call is erroneous.

The \texttt{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 \texttt{C} status argument is undefined.

Two global variables of type \texttt{MPI\_F\_STATUS\_IGNORE} and \texttt{MPI\_F\_STATUSES\_IGNORE} are declared in mpi.h. They can be used to test, in \texttt{C}, whether \texttt{f\_status} is the Fortran value of \texttt{MPI\_STATUS\_IGNORE} or \texttt{MPI\_STATUSES\_IGNORE}, respectively. These are global variables, not \texttt{C} constant expressions and cannot be used in places where \texttt{C} requires constant expressions. Their value is defined only between the calls to \texttt{MPI\_INIT} and \texttt{MPI\_FINALIZE} and should not be changed by user code.

To do the conversion in the other direction, we have the following:

\begin{verbatim}
int MPI\_Status\_c2f(MPI\_Status \*c\_status, MPI\_Fint \*f\_status)
\end{verbatim}

This call converts a \texttt{C} status into a Fortran status, and has a behavior similar to \texttt{MPI\_Status\_f2c}. That is, the value of \texttt{c\_status} must not be either \texttt{MPI\_STATUS\_IGNORE} or \texttt{MPI\_STATUSES\_IGNORE}.

\textbf{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. (\textit{End of advice to users.})

\textbf{Rationale.} The handling of \texttt{MPI\_STATUS\_IGNORE} is required in order to layer libraries with only a \texttt{C} wrapper: if the Fortran call has passed \texttt{MPI\_STATUS\_IGNORE}, then the \texttt{C} wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and \texttt{C}. If \texttt{MPI\_Status\_f2c} were to handle \texttt{MPI\_STATUS\_IGNORE}, then the type of its result would have to be \texttt{MPI\_Status**}, which was considered an inferior solution. (\textit{End of rational.})

\subsection{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.

\textbf{Datatypes} encode the same information in all languages. E.g., a datatype accessor like \texttt{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 same content. If \texttt{f\_status} is the Fortran value of \texttt{MPI\_STATUS\_IGNORE} or \texttt{MPI\_STATUSES\_IGNORE}, or if \texttt{f\_status} is not a valid Fortran status, then the call is erroneous.

The \texttt{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 \texttt{C} status argument is undefined.

Two global variables of type \texttt{MPI\_F\_STATUS\_IGNORE} and \texttt{MPI\_F\_STATUSES\_IGNORE} are declared in mpi.h. They can be used to test, in \texttt{C}, whether \texttt{f\_status} is the Fortran value of \texttt{MPI\_STATUS\_IGNORE} or \texttt{MPI\_STATUSES\_IGNORE}, respectively. These are global variables, not \texttt{C} constant expressions and cannot be used in places where \texttt{C} requires constant expressions. Their value is defined only between the calls to \texttt{MPI\_INIT} and \texttt{MPI\_FINALIZE} and should not be changed by user code.

To do the conversion in the other direction, we have the following:

\begin{verbatim}
int MPI\_Status\_c2f(MPI\_Status \*c\_status, MPI\_Fint \*f\_status)
\end{verbatim}

This call converts a \texttt{C} status into a Fortran status, and has a behavior similar to \texttt{MPI\_Status\_f2c}. That is, the value of \texttt{c\_status} must not be either \texttt{MPI\_STATUS\_IGNORE} or \texttt{MPI\_STATUSES\_IGNORE}.

\textbf{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. (\textit{End of advice to users.})

\textbf{Rationale.} The handling of \texttt{MPI\_STATUS\_IGNORE} is required in order to layer libraries with only a \texttt{C} wrapper: if the Fortran call has passed \texttt{MPI\_STATUS\_IGNORE}, then the \texttt{C} wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and \texttt{C}. If \texttt{MPI\_Status\_f2c} were to handle \texttt{MPI\_STATUS\_IGNORE}, then the type of its result would have to be \texttt{MPI\_Status**}, which was considered an inferior solution. (\textit{End of rational.})

\subsection{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.

\textbf{Datatypes} encode the same information in all languages. E.g., a datatype accessor like \texttt{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
! FORTRAN CODE
REAL R(5)
INTEGER TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) ADDISP(1)
! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GETADDRESS( R, ADDISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, ADDISP, 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...
16.3. LANGUAGE INTEROPERABILITY

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 "stdargs" 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 accessor 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 ≥ 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 "stdargs" 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 accessor 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 ≥ 4 GB. What should be the value returned in Fortran by
4.1.1 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 90,

```
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 of 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 and values can be set in one language and accessed in another. Similarly, attribute keys can be set in one language and accessed in another. To achieve this, attribute keys and values 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_WTIME, etc.)

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_{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.)

MPL_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 90,

```
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 of 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 and values can be set in one language and accessed in another. Similarly, attribute keys can be set in one language and accessed in another. To achieve this, attribute keys and values 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_WTIME, etc.)

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_{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 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 MPI_xxx_GETATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions are used, and an integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if deprecated attribute functions are used.

Example 16.17 A. C to Fortran

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();

The predefined MPI attributes can be integer valued or address valued. Predefined integer valued attributes, such as MPI_TAG_UB, behave as if they were put by a Fortran call, i.e., in Fortran, MPI_COMM_GETATTR(MPI_COMM_WORLD, MPI_TAG_UB, val,
flag, ierr) will return in val the upper bound for tag value; in C,

MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &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,

error) 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

advice 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 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.)
16.3. LANGUAGE INTEROPERABILITY

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 code */
void C_ROUTINE(MPI_Fint *fhandle)
```

Example 16.18 In the example below, a Fortran array is sent from Fortran and received in C.

```c
/* C code */
void C_ROUTINE(MPI_Fint *fhandle)
```
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>
<th>C++ type: const int (or unnamed enum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>MPI::SUCCESS</td>
</tr>
<tr>
<td>MPI_ERR_BUFFER</td>
<td>MPI::ERR_BUFFER</td>
</tr>
<tr>
<td>MPI_ERR_COUNT</td>
<td>MPI::ERR_COUNT</td>
</tr>
<tr>
<td>MPI_ERR_TYPE</td>
<td>MPI::ERR_TYPE</td>
</tr>
<tr>
<td>MPI_ERR_TAG</td>
<td>MPI::ERR_TAG</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
<td>MPI::ERR_COMM</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
<td>MPI::ERR_RANK</td>
</tr>
<tr>
<td>MPI_ERR_REQUEST</td>
<td>MPI::ERR_REQUEST</td>
</tr>
<tr>
<td>MPI_ERR_ROOT</td>
<td>MPI::ERR_ROOT</td>
</tr>
<tr>
<td>MPI_ERR_GROUP</td>
<td>MPI::ERR_GROUP</td>
</tr>
<tr>
<td>MPI_ERR_OP</td>
<td>MPI::ERR_OP</td>
</tr>
<tr>
<td>MPI_ERR_TOPOLOGY</td>
<td>MPI::ERR_TOPOLOGY</td>
</tr>
<tr>
<td>MPI_ERR_DIMS</td>
<td>MPI::ERR_DIMS</td>
</tr>
<tr>
<td>MPI_ERR_ARG</td>
<td>MPI::ERR_ARG</td>
</tr>
<tr>
<td>MPI_ERR_UNKNOWN</td>
<td>MPI::ERR_UNKNOWN</td>
</tr>
<tr>
<td>MPI_ERR_TRUNCATE</td>
<td>MPI::ERR_TRUNCATE</td>
</tr>
<tr>
<td>MPI_ERR_OTHER</td>
<td>MPI::ERR_OTHER</td>
</tr>
<tr>
<td>MPI_ERR_IN_STATUS</td>
<td>MPI::ERR_IN_STATUS</td>
</tr>
</tbody>
</table>

(Continued on next page)
<table>
<thead>
<tr>
<th>Return Codes (continued)</th>
<th>Return Codes (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERR_ACCESS</td>
<td>MPI_ERR_ACCESS</td>
</tr>
<tr>
<td>MPI_ERR_AMODE</td>
<td>MPI_ERR_AMODE</td>
</tr>
<tr>
<td>MPI_ERR_ASSERT</td>
<td>MPI_ERR_ASSERT</td>
</tr>
<tr>
<td>MPI_ERR_BAD_FILE</td>
<td>MPI_ERR_BAD_FILE</td>
</tr>
<tr>
<td>MPI_ERR_BASE</td>
<td>MPI_ERR_BASE</td>
</tr>
<tr>
<td>MPI_ERR_CONVERSION</td>
<td>MPI_ERR_CONVERSION</td>
</tr>
<tr>
<td>MPI_ERR_DISP</td>
<td>MPI_ERR_DISP</td>
</tr>
<tr>
<td>MPI_ERR_DUP_DATAREP</td>
<td>MPI_ERR_DUP_DATAREP</td>
</tr>
<tr>
<td>MPI_ERR_FILE_EXISTS</td>
<td>MPI_ERR_FILE_EXISTS</td>
</tr>
<tr>
<td>MPI_ERR_FILE_IN_USE</td>
<td>MPI_ERR_FILE_IN_USE</td>
</tr>
<tr>
<td>MPI_ERR_FILE</td>
<td>MPI_ERR_FILE</td>
</tr>
<tr>
<td>MPI_ERR_INFO_KEY</td>
<td>MPI_ERR_INFO_VALUE</td>
</tr>
<tr>
<td>MPI_ERR_INFO_NOKEY</td>
<td>MPI_ERR_INFO_NOKEY</td>
</tr>
<tr>
<td>MPI_ERR_INFO_VALUE</td>
<td>MPI_ERR_INFO_KEY</td>
</tr>
<tr>
<td>MPI_ERR_INFO</td>
<td>MPI_ERR_INFO</td>
</tr>
<tr>
<td>MPI_ERR_IO</td>
<td>MPI_ERR_IO</td>
</tr>
<tr>
<td>MPI_ERR_KEYVAL</td>
<td>MPI_ERR_KEYVAL</td>
</tr>
<tr>
<td>MPI_ERR_LOCKTYPE</td>
<td>MPI_ERR_LOCKTYPE</td>
</tr>
<tr>
<td>MPI_ERR_NAME</td>
<td>MPI_ERR_NAME</td>
</tr>
<tr>
<td>MPI_ERR_NO_MEM</td>
<td>MPI_ERR_NO_MEM</td>
</tr>
<tr>
<td>MPI_ERR_NOTSAME</td>
<td>MPI_ERR_NOTSAME</td>
</tr>
<tr>
<td>MPI_ERR_NO_SPACE</td>
<td>MPI_ERR_NO_SPACE</td>
</tr>
<tr>
<td>MPI_ERR_NO_SUCH_FILE</td>
<td>MPI_ERR_NO_SUCH_FILE</td>
</tr>
<tr>
<td>MPI_ERR_PORT</td>
<td>MPI_ERR_PORT</td>
</tr>
<tr>
<td>MPI_ERR_QUOTA</td>
<td>MPI_ERR_QUOTA</td>
</tr>
<tr>
<td>MPI_ERR_READ_ONLY</td>
<td>MPI_ERR_READ_ONLY</td>
</tr>
<tr>
<td>MPI_ERR_RMA_CONFLICT</td>
<td>MPI_ERR_RMA_CONFLICT</td>
</tr>
<tr>
<td>MPI_ERR_RMA_SYNC</td>
<td>MPI_ERR_RMA_SYNC</td>
</tr>
<tr>
<td>MPI_ERR_SERVICE</td>
<td>MPI_ERR_SERVICE</td>
</tr>
<tr>
<td>MPI_ERR_SIZE</td>
<td>MPI_ERR_SIZE</td>
</tr>
<tr>
<td>MPI_ERR_SPAWN</td>
<td>MPI_ERR_SPAWN</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_DATAREP</td>
<td>MPI_ERR_UNSUPPORTED_DATAREP</td>
</tr>
<tr>
<td>MPI_ERR_UNSUPPORTED_OPERATION</td>
<td>MPI_ERR_UNSUPPORTED_OPERATION</td>
</tr>
<tr>
<td>MPI_ERR_WIN</td>
<td>MPI_ERR_WIN</td>
</tr>
<tr>
<td>MPI_ERR_LASTCODE</td>
<td>MPI_ERR_LASTCODE</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_TAG</td>
<td>MPI::ANY_TAG</td>
<td></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>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_STATUS_SIZE</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_TAG</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_ERROR</td>
<td>Not defined for C++</td>
<td></td>
</tr>
</tbody>
</table>

### Variable Address Size (Fortran only)

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ADDRESS_KIND</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_INTEGER_KIND</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_OFFSET_KIND</td>
<td>Not defined for C++</td>
<td></td>
</tr>
</tbody>
</table>

### Error-handling specifiers

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERRORS_ARE_FATAL</td>
<td>MPI::ERRORS_ARE_FATAL</td>
<td></td>
</tr>
<tr>
<td>MPI_ERRORS_RETURN</td>
<td>MPI::ERRORS_RETURN</td>
<td></td>
</tr>
<tr>
<td>MPI_ERRORS_THROW_EXCEPTIONS</td>
<td></td>
<td></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_DATAREP_STRING</td>
<td>MPI::MAX_DATAREP_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>
<tr>
<td>MPI_MAX_PORT_NAME</td>
<td>MPI::MAX_PORT_NAME</td>
<td></td>
</tr>
<tr>
<td>MPI_MAX_PORT_NAME</td>
<td>MPI::MAX_PORT_NAME</td>
<td></td>
</tr>
<tr>
<td>MPI_MAX_PORT_NAME</td>
<td>MPI::MAX_PORT_NAME</td>
<td></td>
</tr>
</tbody>
</table>

### Status size and reserved index values (Fortran only)

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_STATUS_SIZE</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_TAG</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_ERROR</td>
<td>Not defined for C++</td>
<td></td>
</tr>
</tbody>
</table>

### Variable Address Size (Fortran only)

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ADDRESS_KIND</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_INTEGER_KIND</td>
<td>Not defined for C++</td>
<td></td>
</tr>
<tr>
<td>MPI_OFFSET_KIND</td>
<td>Not defined for C++</td>
<td></td>
</tr>
</tbody>
</table>

### Error-handling specifiers

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ name</th>
<th>C++ type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERRORS_ARE_FATAL</td>
<td>MPI::ERRORS_ARE_FATAL</td>
<td></td>
</tr>
<tr>
<td>MPI_ERRORS_RETURN</td>
<td>MPI::ERRORS_RETURN</td>
<td></td>
</tr>
<tr>
<td>MPI_ERRORS_THROW_EXCEPTIONS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Annex A. Language Bindings Summary

#### Named Predefined Datatypes

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>C/C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td></td>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>char</td>
</tr>
<tr>
<td></td>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
<td>signed long long</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>long long (synonym)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td></td>
<td>(treated as integral value)</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned char</td>
</tr>
<tr>
<td></td>
<td>(treated as integral value)</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONGDOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_WCHAR</td>
<td>wchar_t</td>
</tr>
<tr>
<td></td>
<td>(defined in &lt;stddef.h&gt;)</td>
</tr>
<tr>
<td></td>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>(any C/C++ type)</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>(any C/C++ type)</td>
</tr>
</tbody>
</table>

#### C and C++ (no Fortran) Named Predefined Datatypes

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>INTEGER</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>(any Fortran type)</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>(any Fortran type)</td>
</tr>
</tbody>
</table>

#### Fortran types

<table>
<thead>
<tr>
<th>C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>_mpi_char</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>MPI_SHORT</td>
</tr>
<tr>
<td>_mpi_short</td>
</tr>
<tr>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
</tr>
<tr>
<td>_mpi_int</td>
</tr>
<tr>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
</tr>
<tr>
<td>_mpi_long</td>
</tr>
<tr>
<td>signed long</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
</tr>
<tr>
<td>_mpi_signed_char</td>
</tr>
<tr>
<td>_mpi_signed_char</td>
</tr>
<tr>
<td>_mpi_signed_short</td>
</tr>
<tr>
<td>_mpi_signed_int</td>
</tr>
<tr>
<td>_mpi_unsigned_long</td>
</tr>
<tr>
<td>_mpi_unsigned_long_long</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>_mpi_longdouble</td>
</tr>
<tr>
<td>_mpi_wchar</td>
</tr>
<tr>
<td>(defined in &lt;stddef.h&gt;)</td>
</tr>
<tr>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>_mpi_byte</td>
</tr>
<tr>
<td>_mpi_packed</td>
</tr>
</tbody>
</table>

#### Named Predefined Datatypes

<table>
<thead>
<tr>
<th>C++ type: MPI::Datatype</th>
<th>INTEGER</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>(any Fortran type)</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>(any Fortran type)</td>
</tr>
</tbody>
</table>

#### Fortran types

<table>
<thead>
<tr>
<th>C++ types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
</tr>
<tr>
<td>_mpi_char</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>MPI_SHORT</td>
</tr>
<tr>
<td>_mpi_short</td>
</tr>
<tr>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
</tr>
<tr>
<td>_mpi_int</td>
</tr>
<tr>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
</tr>
<tr>
<td>_mpi_long</td>
</tr>
<tr>
<td>signed long</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
</tr>
<tr>
<td>_mpi_signed_char</td>
</tr>
<tr>
<td>_mpi_signed_char</td>
</tr>
<tr>
<td>_mpi_signed_short</td>
</tr>
<tr>
<td>_mpi_signed_int</td>
</tr>
<tr>
<td>_mpi_unsigned_long</td>
</tr>
<tr>
<td>_mpi_unsigned_long_long</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>_mpi_longdouble</td>
</tr>
<tr>
<td>_mpi_wchar</td>
</tr>
<tr>
<td>(defined in &lt;stddef.h&gt;)</td>
</tr>
<tr>
<td>(treated as printable character)</td>
</tr>
<tr>
<td>_mpi_byte</td>
</tr>
<tr>
<td>_mpi_packed</td>
</tr>
</tbody>
</table>
# A.1. DEFINED VALUES AND HANDLES

## C++-Only Named Predefined Datatypes

### C++ types

<table>
<thead>
<tr>
<th>Datatype</th>
<th>C++ Type</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>Datatype</th>
<th>C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DOUBLE_COMPLEX</td>
<td>DOUBLE COMPLEX</td>
</tr>
<tr>
<td>MPI::INTEGER1</td>
<td>INTEGER*1</td>
</tr>
<tr>
<td>MPI::INTEGER2</td>
<td>INTEGER*8</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>Datatype</th>
<th>C++ Type</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::TWOINT</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>Datatype</th>
<th>C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>MPI::DOUBLE_INT</td>
</tr>
<tr>
<td>MPI_INTEGER1</td>
<td>INTEGER*1</td>
</tr>
<tr>
<td>MPI_INTEGER2</td>
<td>INTEGER*8</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>

### Special datatypes for constructing derived datatypes

<table>
<thead>
<tr>
<th>Datatype</th>
<th>C++ Type</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>Datatype</th>
<th>C++ Type</th>
</tr>
</thead>
<tbody>
<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::Comm</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_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

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 (or unnamed enum)</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_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::Comm</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_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

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_IDENT</td>
<td>MPI::IDENT</td>
<td>const int (or unnamed enum)</td>
</tr>
<tr>
<td>MPI_CONGRUENT</td>
<td>MPI::CONGRUENT</td>
<td></td>
</tr>
<tr>
<td>MPI_SIMILAR</td>
<td>MPI::SIMILAR</td>
<td></td>
</tr>
<tr>
<td>MPI_UNEQUAL</td>
<td>MPI::UNEQUAL</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 (or unnamed enum)</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::Comm</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_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
### A.1. DEFINED VALUES AND HANDLES

#### Empty group

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ type</th>
<th>C++ name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_NULL_COPY_FN</td>
<td>const MPI::Group</td>
<td>MPI::EMPTY</td>
</tr>
<tr>
<td>MPI_NULL_DELETE_FN</td>
<td>const MPI::Group</td>
<td>MPI::EMPTY</td>
</tr>
</tbody>
</table>

#### Topologies

<table>
<thead>
<tr>
<th>C/Fortran name</th>
<th>C++ type</th>
<th>C++ name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_GRAPH</td>
<td>const int</td>
<td>MPI::GRAPH</td>
</tr>
<tr>
<td>MPI_CART</td>
<td>const int</td>
<td>MPI::CART</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_APPNUM</td>
<td>MPI::APPNUM</td>
<td>MPI::Copy_function</td>
</tr>
<tr>
<td>MPI_LASTUSEDCODE</td>
<td>MPI::LASTUSEDCODE</td>
<td>MPI::Copy_function</td>
</tr>
<tr>
<td>MPI_UNIVERSE_SIZE</td>
<td>MPI::UNIVERSE_SIZE</td>
<td>MPI::Copy_function</td>
</tr>
<tr>
<td>MPI_WIN_BASE</td>
<td>MPI::WIN_BASE</td>
<td>MPI::Copy_function</td>
</tr>
<tr>
<td>MPI_WIN_DISP_UNIT</td>
<td>MPI::WIN_DISP_UNIT</td>
<td>MPI::Copy_function</td>
</tr>
<tr>
<td>MPI_WIN_SIZE</td>
<td>MPI::WIN_SIZE</td>
<td>MPI::Copy_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>const int</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_LASTUSEDCODE</td>
<td>const int</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_UNIVERSE_SIZE</td>
<td>const int</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_WIN_BASE</td>
<td>const int</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_WIN_DISP_UNIT</td>
<td>const int</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_WIN_SIZE</td>
<td>const int</td>
<td>MPI::Group</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>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_CREATE</td>
<td>MPI::MODE_CREATE</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_DELETE_ON_CLOSE</td>
<td>MPI::MODE_DELETE_ON_CLOSE</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_EXCL</td>
<td>MPI::MODE_EXCL</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_NOCHECK</td>
<td>MPI::MODE_NOCHECK</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_NOPRECEDE</td>
<td>MPI::MODE_NOPRECEDE</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_NOPUT</td>
<td>MPI::MODE_NOPUT</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_NOSTORE</td>
<td>MPI::MODE_NOSTORE</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_NOSUCCEED</td>
<td>MPI::MODE_NOSUCCEED</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_RDONLY</td>
<td>MPI::MODE_RDONLY</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_RDWR</td>
<td>MPI::MODE_RDWR</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_SEQUENTIAL</td>
<td>MPI::MODE_SEQUENTIAL</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_UNIQUE_OPEN</td>
<td>MPI::MODE_UNIQUE_OPEN</td>
<td>MPI::Group</td>
</tr>
<tr>
<td>MPI_MODE_WONLY</td>
<td>MPI::MODE_WONLY</td>
<td>MPI::Group</td>
</tr>
</tbody>
</table>
### Datatype Decoding Constants

<table>
<thead>
<tr>
<th>组合类型</th>
<th>MPI组合类型</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_COMBINER_CONTIGUOUS</td>
<td>MPI_COMBINER_CONTIGUOUS</td>
</tr>
<tr>
<td>MPI_COMBINER_DARRAY</td>
<td>MPI_COMBINER_DARRAY</td>
</tr>
<tr>
<td>MPI_COMBINER_DUP</td>
<td>MPI_COMBINER_DUP</td>
</tr>
<tr>
<td>MPI_COMBINER_F90_COMPLEX</td>
<td>MPI_COMBINER_F90_COMPLEX</td>
</tr>
<tr>
<td>MPI_COMBINER_F90_INTEGER</td>
<td>MPI_COMBINER_F90_INTEGER</td>
</tr>
<tr>
<td>MPI_COMBINER_F90_REAL</td>
<td>MPI_COMBINER_F90_REAL</td>
</tr>
<tr>
<td>MPI_COMBINER_HINDEXED_INTEGER</td>
<td>MPI_COMBINER_HINDEXED_INTEGER</td>
</tr>
<tr>
<td>MPI_COMBINER_HINDEXED</td>
<td>MPI_COMBINER_HINDEXED</td>
</tr>
<tr>
<td>MPI_COMBINER_HVECTOR</td>
<td>MPI_COMBINER_HVECTOR</td>
</tr>
<tr>
<td>MPI_COMBINER_HVECTOR_INTEGER</td>
<td>MPI_COMBINER_HVECTOR_INTEGER</td>
</tr>
<tr>
<td>MPI_COMBINER_INDEXED_BLOCK</td>
<td>MPI_COMBINER_INDEXED_BLOCK</td>
</tr>
<tr>
<td>MPI_COMBINER_INDEXED</td>
<td>MPI_COMBINER_INDEXED</td>
</tr>
<tr>
<td>MPI_COMBINER_NAMED</td>
<td>MPI_COMBINER_NAMED</td>
</tr>
<tr>
<td>MPI_COMBINER_RESIZED</td>
<td>MPI_COMBINER_RESIZED</td>
</tr>
<tr>
<td>MPI_COMBINER_STRUCT_INTEGER</td>
<td>MPI_COMBINER_STRUCT_INTEGER</td>
</tr>
<tr>
<td>MPI_COMBINER_STRUCT</td>
<td>MPI_COMBINER_STRUCT</td>
</tr>
<tr>
<td>MPI_COMBINER_SUBARRAY</td>
<td>MPI_COMBINER_SUBARRAY</td>
</tr>
<tr>
<td>MPI_COMBINER_VECTOR</td>
<td>MPI_COMBINER_VECTOR</td>
</tr>
</tbody>
</table>

### Threads Constants

<table>
<thead>
<tr>
<th>组合类型</th>
<th>MPI组合类型</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_THREAD_FUNNELED</td>
<td>MPI_THREAD_FUNNELED</td>
</tr>
<tr>
<td>MPI_THREAD_MULTIPLIED</td>
<td>MPI_THREAD_MULTIPLIED</td>
</tr>
<tr>
<td>MPI_THREAD_SERIALIZE</td>
<td>MPI_THREAD_SERIALIZE</td>
</tr>
<tr>
<td>MPI_THREAD_SINGLE</td>
<td>MPI_THREAD_SINGLE</td>
</tr>
</tbody>
</table>

### File Operation Constants

<table>
<thead>
<tr>
<th>组合类型</th>
<th>MPI组合类型</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_DISPLACEMENT_CURRENT</td>
<td>MPI_DISPLACEMENT_CURRENT</td>
</tr>
<tr>
<td>MPI_DISTRIBUTION_BLOCK</td>
<td>MPI_DISTRIBUTION_BLOCK</td>
</tr>
<tr>
<td>MPI_DISTRIBUTION_CYCLIC</td>
<td>MPI_DISTRIBUTION_CYCLIC</td>
</tr>
<tr>
<td>MPI_DISTRIBUTION_DFLT</td>
<td>MPI_DISTRIBUTION_DFLT</td>
</tr>
<tr>
<td>MPI_DISTRIBUTION_NONE</td>
<td>MPI_DISTRIBUTION_NONE</td>
</tr>
<tr>
<td>MPI_ORDER_C</td>
<td>MPI_ORDER_C</td>
</tr>
<tr>
<td>MPI_ORDER_FORTRAN</td>
<td>MPI_ORDER_FORTRAN</td>
</tr>
<tr>
<td>MPI_SEARCH_CUR</td>
<td>MPI_SEARCH_CUR</td>
</tr>
<tr>
<td>MPI_SEARCH_CURRENT</td>
<td>MPI_SEARCH_CURRENT</td>
</tr>
</tbody>
</table>

### F90 Datatype Matching Constants

<table>
<thead>
<tr>
<th>组合类型</th>
<th>MPI组合类型</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_TYPECLASS_COMPLEX</td>
<td>MPI_TYPECLASS_COMPLEX</td>
</tr>
<tr>
<td>MPI_TYPECLASS_INTEGER</td>
<td>MPI_TYPECLASS_INTEGER</td>
</tr>
<tr>
<td>MPI_TYPECLASS_REAL</td>
<td>MPI_TYPECLASS_REAL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>组合类型</th>
<th>MPI组合类型</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_TYPECLASS_COMPLEX</td>
<td>MPI_TYPECLASS_COMPLEX</td>
</tr>
<tr>
<td>MPI_TYPECLASS_INTEGER</td>
<td>MPI_TYPECLASS_INTEGER</td>
</tr>
<tr>
<td>MPI_TYPECLASS_REAL</td>
<td>MPI_TYPECLASS_REAL</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 and C++ preprocessor Constants and Fortran Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUBVERSION</td>
</tr>
<tr>
<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 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_Op
MPI_Request
MPI_Win

// C++ opaque types (all within the MPI namespace)

MPI::Aint
MPI::Offset
MPI::Status

Constants Specifying Empty or Ignored Input

<table>
<thead>
<tr>
<th>MPI_ARGV5_NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI::ARGV5_NULL</td>
</tr>
<tr>
<td>MPI_ARGV_NULL</td>
</tr>
<tr>
<td>MPI::ARGV_NULL</td>
</tr>
<tr>
<td>MPI_ERRCODES_IGNORE</td>
</tr>
<tr>
<td>Not defined for C++</td>
</tr>
<tr>
<td>MPI_STATUS_IGNORE</td>
</tr>
<tr>
<td>Not defined for C++</td>
</tr>
</tbody>
</table>

C Constants Specifying Ignored Input (no C++ or Fortran)

<table>
<thead>
<tr>
<th>MPI_F_STATUS_IGNORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not defined for C++</td>
</tr>
</tbody>
</table>

C and C++ preprocessor Constants and Fortran Parameters

<table>
<thead>
<tr>
<th>MPI_SUBVERSION</th>
</tr>
</thead>
<tbody>
<tr>
<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 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_Op
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 *extra_state, void *attribute_val);

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 *extra_state,
     void *attribute_val, void *extra_state);

typedef void MPI_Comm_errhandler_fn(MPI_Comm *, int *, ...);
A.1. DEFINED VALUES AND HANDLES

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_OPCREATE should be declared like this:

SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
  <type> INVEC(LEN), INOUTVEC(LEN)
  INTEGER LEN, TYPE

The copy and delete function arguments to MPI_COMM_KEYVAL_CREATE should be declared like these:

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

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

The copy and delete function arguments to MPI_WIN_KEYVAL_CREATE should be declared like these:

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

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

For Fortran, here are examples of how each of the user-defined subroutines should be declared.

The user-function argument to MPI_OPCREATE should be declared like this:

SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, TYPE)
  <type> INVEC(LEN), INOUTVEC(LEN)
  INTEGER LEN, TYPE

The copy and delete function arguments to MPI_COMM_KEYVAL_CREATE should be declared like these:

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

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

The copy and delete function arguments to MPI_WIN_KEYVAL_CREATE should be declared like these:

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

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

The copy and delete function arguments to MPI_WIN_KEYVAL_CREATE should be declared like these:

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

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

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

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
```

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
```

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
```

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
```

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

SUBROUTINE GREQUEST\_FREE\_FUNCTION(EXTRA\_STATE, IERROR)
  INTEGER IERROR
END\_SUBROUTINE

SUBROUTINE GREQUEST\_CANCEL\_FUNCTION(EXTRA\_STATE, COMPLETE, IERROR)
  INTEGER IERROR
END\_SUBROUTINE
```

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

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
```

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
```

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
```

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
```

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

SUBROUTINE GREQUEST\_FREE\_FUNCTION(EXTRA\_STATE, IERROR)
  INTEGER IERROR
END\_SUBROUTINE

SUBROUTINE GREQUEST\_CANCEL\_FUNCTION(EXTRA\_STATE, COMPLETE, IERROR)
  INTEGER IERROR
END\_SUBROUTINE
```
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 `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 Comm::Errhandler_fn(Comm &, 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);
```

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 `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 Comm::Errhandler_fn(Comm &, 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 de-
clared 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
ture
write_mostly
write_once

file
host
io_node_list
ip_address
ip_port
nb_proc
no_locks
num_io_nodes
path
soft
striping_factor
striping_unit
wdir
false
random
read_mostly
read_once
reverse_sequential
sequential
ture
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_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm, MPI_Request *request)
A.2. C BINDINGS

```c
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,
                      char *datarep, void *inbuf, int incount,
                      )
```
ANNEX A. LANGUAGE BINDINGS SUMMARY

int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)

int MPI_Type_commit(MPI_Datatype *datatype)

int MPI_Type_create_darray(int size, int rank, int ndims, int array_of_gsizes[], int array_of_distribs[], int array_of_dargs[], int array_of_psizes[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_create_hindexed(int count, int array_of_blocklengths[], MPI_Aint array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_create_indexed_block(int count, int blocklength, int array_of_displacements[], MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype)

int MPI_Type_create_struct(int count, int array_of_blocklengths[], MPI_Datatype array_of_types[], MPI_Datatype *newtype)

int MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[], int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_dup(MPI_Datatype type, MPI_Datatype *newtype)

int MPI_Type_free(MPI_Datatype *datatype)

int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers, int max_addresses, int max_datatypes, int array_of_integers[], MPI_Aint array_of_addresses[], MPI_Datatype array_of_datatypes[])

int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers, int *num_addresses, int *num_datatypes, int *combiner)
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 sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoallw(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_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)
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
                MPI_Comm *newintracom)
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
                *attribute_val, 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 Environment 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_call_errhandler(MPI_Comm comm, int errorcode)
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)
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
int MPI_Error_class(int errorcode, int *errorclass)
int MPI_Error_string(int errorcode, char *string, int *resultlen)
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
int MPI_File_create_errhandler(MPI_File_errhandler_fn *function,
              MPI_Errhandler *errhandler)
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
int MPI_Finalized(int *flag)
int MPI_Finalize(void)
int MPI_Free_mem(void *base)
int MPI_Get_processor_name(char *name, int *resultlen)
int MPI_Get_version(int *version, int *subversion)
int MPI_Initialized(int *flag)
int MPI_Init(int *argc, char ***argv)
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
int MPI_Win_create_errhandler(MPI_Win_errhandler_fn *function,
              MPI_Errhandler *errhandler)
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
double MPI_Wtick(void)
double MPI_Wtime(void)

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_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
int MPI_Error_class(int errorcode, int *errorclass)
int MPI_Error_string(int errorcode, char *string, int *resultlen)
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
int MPI_File_create_errhandler(MPI_File_errhandler_fn *function,
              MPI_Errhandler *errhandler)
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
int MPI_Finalized(int *flag)
int MPI_Finalize(void)
int MPI_Free_mem(void *base)
int MPI_Get_processor_name(char *name, int *resultlen)
int MPI_Get_version(int *version, int *subversion)
int MPI_Initialized(int *flag)
int MPI_Init(int *argc, char ***argv)
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
int MPI_Win_create_errhandler(MPI_Win_errhandler_fn *function,
              MPI_Errhandler *errhandler)
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
double MPI_Wtick(void)
double MPI_Wtime(void)

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)[*], 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_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)[*], 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)
A.2. C Bindings

```c
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(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

```c
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_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_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_Type_c2f(MPI_Datatype datatype)
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Type_c2f(MPI_Type datatype)
MPI_Type MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Win_c2f(MPI_Win win)
MPI_Win MPI_Win_f2c(MPI_Fint win)
MPI_Fint MPI_Win_c2f(MPI_Win win)
MPI_Win MPI_Win_f2c(MPI_Fint win)
```

A.2 C BINDINGS

```c
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

```c
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_COMM_c2f(MPI_COMM comm)
MPI_Comm MPI_COMM_f2c(MPI_Fint comm)
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_Type_c2f(MPI_Datatype datatype)
MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Type_c2f(MPI_Type datatype)
MPI_Type MPI_Type_f2c(MPI_Fint datatype)
MPI_Fint MPI_Win_c2f(MPI_Win win)
MPI_Win MPI_Win_f2c(MPI_Fint win)
MPI_Fint MPI_Win_c2f(MPI_Win win)
MPI_Win MPI_Win_f2c(MPI_Fint win)
```
A.2.13 Profiling Interface C Bindings

```c
int MPI_Pcontrol(const int level, ...)
```

A.2.14 Deprecated C Bindings

```c
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)
  \textit{type} BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

MPI_BSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  \textit{type} BUF(*)
  INTEGER REQUEST, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_BUFFER_ATTACH(BUFFER, SIZE, IERROR)
  \textit{type} BUFFER(*)
  INTEGER SIZE, IERROR

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
  \textit{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)
  \textit{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)
  \textit{type} BUF(*)
  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_IRSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  \textit{type} BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  \textit{type} BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI_ISSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
  \textit{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)
  \textit{type} BUF(*)
  INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR
ANNEX A. LANGUAGE BINDINGS SUMMARY

1. `<type> BUF(*)
2. INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
3. MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
4. `<type> BUF(*)
5. INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
6. MPI_RECV(REQUEST, IERROR)
7. INTEGER REQUEST, IERROR
8. MPI_RECV_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
9. INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR, LOGICAL FLAG
10. MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
11. `<type> BUF(*)
12. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
13. MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
14. `<type> Buf(*)
15. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
16. MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
17. `<type> Buf(*)
18. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
19. MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
20. `<type> Buf(*)
21. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
22. MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)
23. `<type> Buf(*)
24. INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR
25. MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR)
26. `<type> Sendbuf(*), Recvbuf(*)
27. INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
28. MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
29. `<type> Buf(*)
30. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
31. MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
32. `<type> Buf(*)
33. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
34. MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
35. INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
36. MPI_REQUEST_FREE(REQUEST, IERROR)
37. INTEGER REQUEST, IERROR
38. MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
39. INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR, LOGICAL FLAG
40. MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
41. `<type> Buf(*)
42. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
43. MPI_RSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
44. `<type> Buf(*)
45. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
46. MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
47. `<type> Buf(*)
48. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
49. MPI_SEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
50. `<type> Buf(*)
51. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
52. MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)
53. `<type> Buf(*)
54. INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR
55. MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR)
56. `<type> Sendbuf(*), Recvbuf(*)
57. INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
58. MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
59. `<type> Buf(*)
60. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
61. MPI_SSEND_INIT(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
62. `<type> Buf(*)
63. INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
64. MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
65. 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)
  INTEGER Status(MPI_STATUS_SIZE), IERROR

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
  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(*), 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)
  <type> 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)
  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)
  INTEGER Status(MPI_STATUS_SIZE), IERROR

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
  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)
  <type> 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 <type> INBUF(*), OUTBUF(*)

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)
<type> INBUF(*), OUTBUF(*) 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_DISTRIBS, ARRAY_OF_PSIZES, ORDER, OLDTYPE, NEWTYPE, IERROR)
INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
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(*), OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)

MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE

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_TYPES(*), NEWTYPE,
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,
ANNEX A. LANGUAGE BINDINGS SUMMARY

IERROR)
INTEGER INBUF(*), OUTBUF(*)

A.3.3 Collective Communication Fortran Bindings

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPF,
COMM, IERROR)
INTEGER SENDBUF(*), RECVBUF(*)

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUFF, RECVCOUNTS, DISPLS,
RECVTYPE, COMM, IERROR)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
IERROR

MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
INTEGER SENDBUF(*), RECVBUF(*)

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUFF, RECVCOUNT, RECVTYPE,
COMM, IERROR)
INTEGER SENDBUF(*), RECVBUF(*)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUFF, RECVCOUNTS,
RDISPLS, RECVTYPE, COMM, IERROR)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
RECVTYPE, COMM, IERROR

MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUFF, RECVCOUNTS,
RDISPLS, RECVTYPES, COMM, IERROR)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
RDISPLS(*), RECVTYPES(*), COMM, IERROR

MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
INTEGER BUFFER(*)

MPI_EXSCAN(SENDBUF, RECVBUFF, COUNT, DATATYPE, OP, COMM, IERROR)
INTEGER SENDBUF(*), RECVBUFF(*)

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUFF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)
INTEGER SENDBUF(*), RECVBUFF(*)
A.3. FORTRAN BINDINGS

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
RECVTYPE, ROOT, COMM, IERROR)

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
COMM, IERROR

MPI_OP_CREATE( FUNCTION, COMMUTE, OP, IERROR)
EXTERNAL FUNCTION

LOGICAL COMMUTE

INTEGER OP, IERROR

MPI_OP_FREE( OP, IERROR)

INTEGER OP, IERROR

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
IERROR)

INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)

INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)

INTEGER COUNT, DATATYPE, OP, COMM, IERROR

MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
ROOT, COMM, IERROR)

INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
RECVTYPE, ROOT, COMM, IERROR)

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

A.3. FORTRAN BINDINGS
ANNEX A. LANGUAGE BINDINGS SUMMARY

```
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 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_SETATTR(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)

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_SETATTR(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_SETATTR(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
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_CARTDIM_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, NDIMS, DIMS, IERROR)
INTEGER COMM, 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_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH, IERROR)
INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
LOGICAL REORDER

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

MPI_FILE_ERROR_CLASS(FILE, ERRORCLASS, IERROR)
   INTEGER FILE, ERRORCLASS, IERROR

MPI_FILE_ERROR_STRING(FILE, ERRORCODE, STRING, RESULTLEN, IERROR)
   INTEGER FILE, ERRORCODE, STRING, RESULTLEN, IERROR
   CHARACTER*(*) STRING

MPI_FILE_ERRHANDLER_FREE(FILE, ERRHANDLER, IERROR)
   INTEGER FILE, ERRHANDLER, IERROR
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)

---

MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
INTEGER FILE, ERRHANDLER, IERROR

MPI_FINALIZED(FLAG, IERROR)
LOGICAL FLAG
INTEGER IERROR

MPI_FINALIZE(IERROR)
INTEGER IERROR

MPI_FREE_MEM(BASE, IERROR)
$type$ BASE(*) INTEGER IERROR

MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
CHARACTER*(*) NAME
INTEGER RESULTLEN, IERROR

MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
INTEGER VERSION, SUBVERSION, IERROR

MPI_INITIALIZED(FLAG, IERROR)
LOGICAL FLAG
INTEGER IERROR

MPI_INIT(IERROR)
INTEGER IERROR

MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
INTEGER WIN, ERRORCODE, IERROR

MPI_WIN_CREATE_ERRHANDLER(FUNCTION, ERRHANDLER, IERROR)
EXTERNAL FUNCTION INTEGER ERRHANDLER, IERROR

MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
INTEGER WIN, ERRHANDLER, IERROR

MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
INTEGER WIN, ERRHANDLER, IERROR

---

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)
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_SPAWNCOMMAND, 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,
ANNEX A. LANGUAGE BINDINGS SUMMARY

ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES, IERROR
INTEGER COUNT, ARRAY_OF_INFO(*), ARRAY_OF_MAXPROCS(*), ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
CHARACTER(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(COUNT, *)

MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
CHARACTER(*) SERVICE_NAME, PORT_NAME INTEGER INFO, IERROR

MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
CHARACTER(*) PORT_NAME INTEGER INFO, IERROR

MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
INTEGER INFO, IERROR CHARACTER(*) SERVICE_NAME, PORT_NAME

MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
INTEGER INFO, IERROR CHARACTER(*) SERVICE_NAME, PORT_NAME

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)
	[type] ORIGIN_ADDR(*) 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)
	[type] ORIGIN_ADDR(*) 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)
	[type] ORIGIN_ADDR(*) 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)
	[type] BASE(*) 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, IERROR LOGICAL 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, IERROR EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN INTEGER (KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
INTEGER REQUIRED, PROVIDED, IERROR

MPI_IS_THREAD_MAIN(FLAG, IERROR)
INTEGER FLAG
LOGICAL IERROR

MPI_QUERY_THREAD(PROVIDED, IERROR)
INTEGER PROVIDED
LOGICAL IERROR

MPI_STATUS_SET_CANCELLEDSTATUS, FLAG, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), IERROR LOGICAL 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, ETYP, FILETYPE, DATAREP, IERROR)
  INTEGER FH, ETYP, 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)
  <type> BUF(*)
  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)
  <type> BUF(*)
  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
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, IERROR
MPI_FILE_WRITE.Ordered_END(FH, BUF, STATUS, IERROR)
  <type> BUF(*) INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), 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)
ANNEX A. LANGUAGE BINDINGS SUMMARY

A.3.12 Language Bindings Fortran Bindings

MPI_SIZEOF(X, SIZE, IERROR)
  Integer X, 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)
  Integer LOCATION(*), 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

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

MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, 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)

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
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
    Prequest 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& ... 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
    Prequest Comm::Recv_init(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) 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(const void *sendbuf, int sendcount, const
                     Datatype& sendtype, int dest, int sendtag, void *recvbuf,
                     int recvcount, const Datatype& recvtype, int source,
                     int recvtag, Status& status) const

void Comm::Sendrecv_replace(void* buf, int count, const
                            Datatype& datatype, int dest, int sendtag, int source,
                            int recvtag) const

void Comm::Sendrecv_replace(void* buf, int count, const
                            Datatype& datatype, int dest, int sendtag, int source,
                            int recvtag, 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[])
A.4. C++ BINDINGS

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[]) const

};

A.4. C++ BINDINGS

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[]) 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
void Datatype::Dup() 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
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
void Datatype::Dup() 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
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 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 recvcounts[],
const int displs[], constDatatype& 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
    Intracomm 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)
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)
bool Comm::Is_inter() const
Intracomm Intercomm::Merge(bool high) const
Group Group::Range_excl(int n, const int ranges[][3]) const
Group Group::Range_incl(int n, const int ranges[][3]) const
void Comm::Set_attr(int comm_keyval, const void* attribute_val) const
void Datatype::Set_attr(int type_keyval, const void* attribute_val)
void Win::Set_attr(int win_keyval, const void* attribute_val)
void Comm::Set_name(const char* comm_name)
void Datatype::Set_name(const char* type_name)
void Win::Set_name(const char* win_name)
Intracomm Intercomm::Split(int color, int key) const
static void Group::Translate_ranks (const Group& group1, int n,
const int ranks1[], const Group& group2, int ranks2[])
static Group Group::Union(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

bool Comm::Is_inter() const
Intracomm Intercomm::Merge(bool high) const
Group Group::Range_excl(int n, const int ranges[][3]) const
Group Group::Range_incl(int n, const int ranges[][3]) const
void Comm::Set_attr(int comm_keyval, const void* attribute_val) const
void Datatype::Set_attr(int type_keyval, const void* attribute_val)
void Win::Set_attr(int win_keyval, const void* attribute_val)
void Comm::Set_name(const char* comm_name)
void Datatype::Set_name(const char* type_name)
void Win::Set_name(const char* win_name)
Intracomm Intercomm::Split(int color, int key) const
static void Group::Translate_ranks (const Group& group1, int n,
const int ranks1[], const Group& group2, int ranks2[])
static Group Group::Union(const Group& group1, const Group& group2)
A.4. C++ BINDINGS

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 Environmenta Management C++ Bindings

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(Aint 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)
}
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

Intercomm Intracomm::Connect(const char* port_name, const Info& info, int root) const
void Comm::Disconnect() static
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, 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[] ) const
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[])
void Unpublish_name(const char* service_name, const Info& info, const char* port_name)

A.4.9 One-Sided Communications C++ Bindings

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

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
void Win::Free()
void Win::Get_group() const
void Win::Get(void *origin_addr, int origin_count, const Datatype& origin_datatype, int target_rank, int 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, int 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 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. C++ BINDINGS

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::Iread(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(void* buf, int count, const Datatype& datatype)
void File::Read_all(void* buf, int count, const Datatype& datatype, Status& status)
void File::Read_all(void* buf, int count, const Datatype& datatype, Status& status, 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(void* buf, int count,
const Datatype& datatype)
void File::Read_at_all(Offset offset, void* buf, int count,
const Datatype& datatype, Status& status)
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(void* buf, int count, const Datatype& datatype,
Status& status)
void File::Read_shared(void* buf, int count, const Datatype& datatype,
Status& status)
void File::Read(void* buf, int count, const Datatype& datatype, Status& status)
void Register_datarep(const char* datarep,
Datatype_conversion_function* read_conversion_fn,
Datatype_conversion_function* write_conversion_fn,
Datatype_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)
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)
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)
void File::Write_shared(const void* buf, int count, const Datatype& datatype, Status& status)
void File::Write_shared(const void* buf, int count, const Datatype& datatype)

};

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. C++ BINDINGS

A.4.16 Construction / Destruction
namespace MPI {
    ⟨CLASS⟩::⟨CLASS⟩()
    ⟨CLASS⟩::~⟨CLASS⟩()
};

A.4.17 Copy / Assignment
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.
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.
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).
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 when a keyval that was created by a call to MPI XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI YYYY_GET_ATTR, MPI YYYY_SET_ATTR, MPI YYYY_DELETE_ATTR, or MPI YYYY_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 then MPI_MAX_OBJECT-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then 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: If 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 when a keyval that was created by a call to MPI XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI YYYY_GET_ATTR, MPI YYYY_SET_ATTR, MPI YYYY_DELETE_ATTR, or MPI YYYY_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 then MPI_MAX_OBJECT-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then 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: If 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 than MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger than 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_NTHKEY, MPI_INFO_GET_NTHKEY, 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_SET_INFO: 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.

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_MODE_SEQUENTIAL, 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_xxxx:

   **Advice to implementors.** An application may often repeat a call to MPI_TYPE_CREATE_F90_xxxx with the same combination of (xxx, p, r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same (xxx, p, r) combination.
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


BIBLIOGRAPHY


[34] nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2


[37] 4.4BSD Programmer’s Supplementary Documents (PSD). O’Reilly and Associates, 1994. 10.5.5


BIBLIOGRAPHY


[34] nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990. 1.2


[37] 4.4BSD Programmer’s Supplementary Documents (PSD). O’Reilly and Associates, 1994. 10.5.5


1.2


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
C/Fortran handle conversion, 480
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 - untyped, 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
MPI_ACCUMULATE, 332
MPI_ADDRESS, 95
MPI_Addres, 114, 117, 118, 124
MPI_Aint, 114
MPI_Allgather, 154
MPI_ALLOC_MEM, 263
MPI_ALLOC_mem, 264
MPI_ALLREDUCE, 172
MPI_Barrier, 280, 281, 347
MPI_Bcast, 136, 177, 178
MPI_Bsend, 42
MPI_Buffer_attach, 46, 280
MPI_Buffer_detach, 46
MPI_BYTE, 35
MPI_Cancel, 281
MPI_CART_COORDS, 253
MPI_CART_GET, 256
MPI_CART_RANK, 256
MPI_CART_SHIFT, 253
MPI_CART_SUB, 254
MPI_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.

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
C/Fortran handle conversion, 480
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 - untyped, 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
MPI_ACCUMULATE, 332
MPI_ADDRESS, 95
MPI_Addres, 114, 117, 118, 124
MPI_Aint, 114
MPI_Allgather, 154
MPI_ALLOC_MEM, 263
MPI_ALLOC_mem, 264
MPI_ALLREDUCE, 172
MPI_Barrier, 280, 281, 347
MPI_Bcast, 136, 177, 178
MPI_Bsend, 42
MPI_Buffer_attach, 46, 280
MPI_Buffer_detach, 46
MPI_BYTE, 35
MPI_Cancel, 281
MPI_CART_COORDS, 253
MPI_CART_GET, 256
MPI_CART_RANK, 256
MPI_CART_SHIFT, 253
MPI_CART_SUB, 254
MPI_CHARACTER, 36
<table>
<thead>
<tr>
<th>Examples Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_start, 339, 347</td>
</tr>
<tr>
<td>MPI_Win_unlock, 344</td>
</tr>
<tr>
<td>MPI_Win_wait, 347</td>
</tr>
<tr>
<td>mpiexec, 286</td>
</tr>
<tr>
<td>Non-deterministic program with MPI_Bcast, 178</td>
</tr>
<tr>
<td>Non-overtaking messages, 42</td>
</tr>
<tr>
<td>Nonblocking operations, 54, 55</td>
</tr>
<tr>
<td>Nonblocking operations - message ordering, 56</td>
</tr>
<tr>
<td>Nonblocking operations - progress, 56</td>
</tr>
<tr>
<td>Threads and MPI, 366</td>
</tr>
<tr>
<td>Typemap, 79–81, 83, 86, 93</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_start, 339, 347</td>
</tr>
<tr>
<td>MPI_Win_unlock, 344</td>
</tr>
<tr>
<td>MPI_Win_wait, 347</td>
</tr>
<tr>
<td>mpiexec, 286</td>
</tr>
<tr>
<td>Non-deterministic program with MPI_Bcast, 178</td>
</tr>
<tr>
<td>Non-overtaking messages, 42</td>
</tr>
<tr>
<td>Nonblocking operations, 54, 55</td>
</tr>
<tr>
<td>Nonblocking operations - message ordering, 56</td>
</tr>
<tr>
<td>Nonblocking operations - progress, 56</td>
</tr>
<tr>
<td>Threads and MPI, 366</td>
</tr>
<tr>
<td>Typemap, 79–81, 83, 86, 93</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>Page Numbers</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:CHARACTER</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_RETURN</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>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:INTEGER8S</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LOGICAL MRI:BOOL</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LONG</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE_COMPLEX</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LONG_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LONG_LONG</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>455</td>
</tr>
<tr>
<td>MPI:REAL4</td>
<td>455</td>
</tr>
<tr>
<td>MPI:REALS</td>
<td>455</td>
</tr>
<tr>
<td>MPI:SHORT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SHORT_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:SIGN char</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SIGN char, MPI:UNSIGNED_CHAR</td>
<td>454</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_RETURN</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>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:INTEGER8S</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LOGICAL MRI:BOOL</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LONG</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE_COMPLEX</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LONG_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LONG_LONG</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>455</td>
</tr>
<tr>
<td>MPI:REAL4</td>
<td>455</td>
</tr>
<tr>
<td>MPI:REALS</td>
<td>455</td>
</tr>
<tr>
<td>MPI:SHORT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SHORT_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:SIGN char</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SIGN char, MPI:UNSIGNED_CHAR</td>
<td>454</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_RETURN</td>
<td>19</td>
</tr>
<tr>
<td>MPI:ERRORS_THROW.Exceptions</td>
<td>19, 23, 265</td>
</tr>
</tbody>
</table>

---

This index lists predefined MPI constants and handles.

<table>
<thead>
<tr>
<th>Constant/Handle</th>
<th>Page Numbers</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:CHARACTER</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_RETURN</td>
<td>19</td>
</tr>
<tr>
<td>MPI:ERRORS.ThrowExceptions</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>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:INTEGER8S</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LOGICAL MRI:BOOL</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LOGICAL</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LONG</td>
<td>454</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE</td>
<td>453, 454</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE_COMPLEX</td>
<td>454, 495</td>
</tr>
<tr>
<td>MPI:LONG_DOUBLE_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LONG_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:LONG_LONG</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>455</td>
</tr>
<tr>
<td>MPI:REAL4</td>
<td>455</td>
</tr>
<tr>
<td>MPI:REALS</td>
<td>455</td>
</tr>
<tr>
<td>MPI:SHORT</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SHORT_INT</td>
<td>455</td>
</tr>
<tr>
<td>MPI:SIGN char</td>
<td>454</td>
</tr>
<tr>
<td>MPI:SIGN char, MPI:UNSIGNED_CHAR</td>
<td>454</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_RETURN</td>
<td>19</td>
</tr>
<tr>
<td>MPI:ERRORS.ThrowExceptions</td>
<td>19, 23, 265</td>
</tr>
</tbody>
</table>

---

This index lists predefined MPI constants and handles.
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, 19, 29, 39, 41, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 54, 56, 64, 67, 69, 72, 358, 361, 393, 397, 398, 401, 450, 480
MPI::Address, 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, 323, 324, 326, 328, 331, 338-343, 450, 480
MPI::Aint, 15, 19, 29, 39, 41, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 54, 56, 64, 67, 69, 72, 358, 361, 393, 397, 398, 401, 450, 480
MPI::Comm, 26, 188, 193-196, 198, 201, 210, 211, 213, 223, 226, 227, 450, 456, 466
MPI::Datatype, 70, 450
MPI::Errhandler, 266, 267-270, 446, 447, 450, 480
MPI::Exception, 450, 457
MPI::Graphcomm, 246, 450, 456
MPI::Grequest, 358, 428, 450
MPI::Group, 186, 198, 187-192, 196, 211, 324, 329, 340, 380, 450, 480
MPI::Info, 262, 287, 287, 288-291, 296, 299, 301, 308-312, 375, 378, 382, 383, 385, 450, 480
MPI::Intercomm, 450, 456
MPI::Intraconn, 450, 456
MPI::Offset, 15, 19, 379, 380, 385, 387, 391-393, 398, 399, 403, 404, 406, 407, 417, 424
MPI:Op, 160, 168, 172-175, 331, 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

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, 19, 29, 39, 41, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 54, 56, 64, 67, 69, 72, 358, 361, 393, 397, 398, 401, 450, 480
MPI::Win, 228-230, 238, 268, 276, 322, 323, 324, 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, 466
MPI::Datatype, 70, 450
MPI::Errhandler, 266, 267-270, 446, 447, 450, 480
MPI::Exception, 450, 457
MPI::Graphcomm, 246, 450, 456
MPI::Grequest, 358, 428, 450
MPI::Group, 186, 198, 187-192, 196, 211, 324, 329, 340, 380, 450, 480
MPI::Info, 262, 287, 287, 288-291, 296, 299, 301, 308-312, 375, 378, 382, 383, 385, 450, 480
MPI::Intercomm, 450, 456
MPI::Intraconn, 450, 456
MPI::Offset, 15, 19, 379, 380, 385, 387, 391-393, 398, 399, 403, 404, 406, 407, 417, 424, 478
MPI:Op, 160, 168, 170-175, 331, 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

MPI_Win, 228–230, 238, 268, 276, 322, 323, 324, 326, 328, 331, 338–343, 480

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48

MPI_Declarations_Index

MPI_Win, 228–230, 238, 268, 276, 322, 323, 324, 326, 328, 331, 338–343, 480

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
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
MPI Function Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_TEST_CANCELLD, 52-54, 68, 68, 359, 365, 390</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TESTALL, 57, 60, 60, 359, 360, 363, 366</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_ANY, 52, 57, 58, 58, 62, 359, 360, 363, 366</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TSTESOME, 57, 61, 62, 359, 360, 363, 366</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TOOPO_TEST, 243, 248, 248</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_C2F, 490</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_COMMIT, 99, 99, 480</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CONTIGUOUS, 12, 79, 79, 81, 96, 106, 374, 413</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_DARRAY, 12, 32, 90, 90, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_F90_COMPLEX, 12, 106, 108, 415, 469, 472, 473, 474</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_F90_INTEGER, 12, 106, 108, 415, 469, 472, 473, 474</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_F90_REAL, 12, 106, 108, 415, 469, 472, 474</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 84, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_BLOCK, 12, 85, 106</td>
<td>585</td>
</tr>
<tr>
<td>MPI_TYPE_CREATE_INDEXED_VECTOR, 12, 17, 79, 81, 81, 106, 441</td>
<td>585</td>
</tr>
</tbody>
</table>
MPI_WIN_CREATE_KEYVAL, 222, 228, 233, 486, 563
MPI_WIN_DELETE_ATTR, 222, 229, 233
MPI_WIN_DUP_FN, 228, 228
MPI_WIN_F2C, 340
MPI_WIN_FENCE, 324, 334, 338, 338, 344, 349–352
MPI_WIN_FREE, 229, 319, 322, 324
MPI_WIN_FREE_KEYVAL, 222, 229, 233
MPI_WIN_GET_ATTR, 222, 230, 233, 234, 348
MPI_WIN_GET_ERRHANDLER, 265, 268, 564
MPI_WIN_GET_GROUP, 224, 235
MPI_WIN_GET_NAME, 238
MPI_WIN_KEYVAL_CREATE, 501
MPI_WIN_LOCK, 202, 233, 234, 242, 243, 344, 359
MPI_WIN_NULL_COPY_FN, 228, 228
MPI_WIN_NULL_DELETE_FN, 228
MPI_WIN_POST, 324, 234, 339, 340, 340, 341–345, 352
MPI_WIN_POST, MPI_WIN_FENCE, 350
MPI_WIN_SET_ATTR, 222, 229, 233
MPI_WIN_SET_ERRHANDLER, 265, 268
MPI_WIN_SET_NAME, 238
MPI_WIN_START, 334, 339, 340, 340, 342, 344, 345, 348
MPI_WIN_TEST, 341, 341
MPI_WIN_UNLOCK, 202, 233, 234, 243, 344, 349, 350, 352
MPI_WIN_WAIT, 324, 334, 340, 340, 341–343, 350, 352
MPI_WTICK, 21, 278, 278
MPI_WTIME, 21, 261, 277, 277, 278
mpirun, 279, 283, 285, 285, 300
pmiprun, 284, 285
PMPI_, 345
PMPI_WTICK, 21
PMPI_WTIME, 21