# MPI: A Message-Passing Interface Standard

Version 4.0

(Draft)

Unofficial, for comment only

Message Passing Interface Forum

November 26, 2019

This document describes a draft version of the Message-Passing Interface (MPI) standard, version 4.0, intended for comment. It is not an official version of the standard. 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 and Fortran are defined.

Historically, the evolution of the standards is from MPI-1.0 (May 5, 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 to MPI-2.1 (June 23, 2008), combining the previous documents. Version MPI-2.2 (September 4, 2009) added additional clarifications and seven new routines. Version MPI-3.0 (September 21, 2012) is an extension of MPI-2.2. Version MPI-3.1 (June 4, 2015) adds clarifications and minor extensions to MPI-3.0.

Comments. Please send comments on MPI to the MPI Forum as follows:

1. Subscribe to `http://lists.mpi-forum.org/mailman/listinfo.cgi/mpi-comments`

2. Send your comment to: mpi-comments@mpi-forum.org, together with the URL of the version of the MPI standard and the page and line numbers on which you are commenting.

   Your comment will be forwarded to MPI Forum committee members for consideration. Messages sent from an unsubscribed e-mail address will not be considered.

Version 4.0: XXXX 2020.

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

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

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

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

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

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

Version 1.2: July 18, 1997. The MPI-2 Forum introduced MPI-1.2 as Chapter 3 in the standard "MPI-2: Extensions to the Message-Passing Interface", July 18, 1997. This section 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. The changes from Version 1.0 are minor. A version of this document with all changes marked is available.

**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.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

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

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 Communication
- Al Geist, Marc Snir, Steve Otto, Collective Communication
- 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 | Rob 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 Rigsbee | Arch Robison |
| Erich Schikuta | Ambuj Singh | Alan Sussman | Robert Tomlinson |
| Robert G. Voigt | Dennis Weeks | Stephen Wheat | Steve Zenith |

The University of Tennessee and Oak Ridge National Laboratory made the draft available by anonymous FTP mail servers and were instrumental in distributing the document.

The work on the MPI-1 standard was supported in part by ARPA and NSF under grant ASC-9310330, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and by the Commission of the European Community through Esprit project P6643 (PPPE).

### 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 | Pang Chen |
| Ying Chen | Albert Cheng | Yong Cho | Joel Clark |
| Lyndon Clarke | Laurie Costello | Dennis Cottel | Jim Cownie |
| Zhenqian Cui | Suresh Damodaran-Kamal | | Raja Daoud |
| Judith Devaney | David DiNucci | Doug Doefler | Jack Dongarra |
| Terry Dontje | Nathan Doss | Anne Elster | Mark Fallon |
| Karl 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 | Joefon 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 |
| John May | Oliver McBryan | Brian McCandless | Tyce McLarty |
| Thom McMahon | Harish Nag | Nick Nevin | Jarek Nieplocha |
| Ron Oldfield | Peter Ossadnik | Steve Otto | Peter Pacheco |
| Yoonho Park | Perry Partow | Pratap Pattnaik | Elsie Pierce |
| Paul Pierce | Heidi Poxon | Jean-Pierre Prost | Boris Protopopov |
| James Pruyve | Rolf Rabenseifner | Joe Rieken | Peter Rigsbee |
| Tom Robey | Anna Rounbehler | Nobutoshi Sagawa | Arindam Saha |
| Eric Salo | Darren Sanders | Eric Sharakan | Andrew Sherman |
| Fred Shirley | Lance Shuler | A. Gordon Smith | Ian Stockdale |
| David Taylor | Stephen Taylor | Greg Tensa | Rajeev Thakur |
| Marydell Tholburn | Dick Treumann | Simon Tsang | Manuel Ujaldon |
| David Walker | Jerrell Watts | Klaus Wolf | Parkson Wong |
| Dave Wright | | | |

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 Techology, 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 Adminstration
Oak Ridge National Laboratory
The Ohio State University
PALLAS GmbH
Pacific Northwest National Laboratory
Pratt & Whitney
San Diego Supercomputer Center
Sanders, A Lockheed-Martin Company
Sandia National Laboratories
Schlumberger
Scientific Computing Associates, Inc.
Silicon Graphics Incorporated
Sky Computers
Sun Microsystems Computer Corporation
Syracuse University
The MITRE Corporation
Thinking Machines Corporation
United States Navy
University of Colorado
University of Denver
University of Houston
University of Illinois
University of Maryland
University of Notre Dame
University of San Fransisco
University of Stuttgart Computing Center
University of Wisconsin

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)
- Rolf Rabenseifner, Steering Committee, Merge of MPI-2.1 and MPI-2.1-Errata Ballots 3, 4 (2008)

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, Front matter, Introduction, and Bibliography
- Richard Graham, Point-to-Point Communication
- 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 R. de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O
- Jeffrey M. Squyres, Language Bindings and MPI-2.1 Secretary
- Rolf Rabenseifner, Deprecated Functions and Annex Change-Log
- Alexander Supalov and Denis Nagorny, 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 Dontje | Gabor Dozsa | 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 Kalemkarian |
| 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 Tipparaju |
| 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
Cisco Systems, Inc.
Cray Inc.
The HDF Group
Hewlett-Packard
IBM T.J. Watson Research
Indiana University
Institut National de Recherche en Informatique et Automatique (INRIA)
Intel Corporation
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
Mathworks
Mellanox Technologies
Microsoft
Myricom
NEC Laboratories Europe, NEC Europe Ltd.
Oak Ridge National Laboratory
The Ohio State University
Pacific Northwest National Laboratory
QLogic Corporation
Sandia National Laboratories
SiCortex
Silicon Graphics Incorporated
Sun Microsystems, Inc.
University of Alabama at Birmingham
University of Houston

University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Wisconsin

Funding for the MPI Forum meetings was partially supported by award #CCF-0816909 from the National Science Foundation. In addition, the HDF Group provided travel support for one U.S. academic.

## MPI-2.2:

All chapters have been revisited to achieve a consistent MPI-2.2 text. Those who served as authors for the necessary modifications are:

- William Gropp, Front matter, Introduction, and Bibliography; MPI-2.2 chair.
- Richard Graham, Point-to-Point Communication and Datatypes
- Adam Moody, Collective Communication
- Torsten Hoefler, Collective Communication and Process Topologies
- Richard Treumann, Groups, Contexts, and Communicators
- Jesper Larsson Träff, Process Topologies, Info-Object and One-Sided Communications
- George Bosilca, Datatypes and Environmental Management
- David Solt, Process Creation and Management
- Bronis R. de Supinski, External Interfaces, and Profiling
- Rajeev Thakur, I/O
- Jeffrey M. Squyres, Language Bindings and MPI-2.2 Secretary
- Rolf Rabenseifner, Deprecated Functions, Annex Change-Log, and Annex Language Bindings
- Alexander Supalov, 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 | Greg Bronevetsky |
| Jeff Brown | Darius Buntinas | Jonathan Carter |
| Nathan DeBardeleben | Terry Dontje | Gabor Dozsa |
| Edric Ellis | Karl Feind | Edgar Gabriel |
| Patrick Geoffray | Johann George | David Gingold |
| David Goodell | Erez Haba | Robert Harrison |
| Thomas Herault | Marc-André Hermanns | Steve Hodson |
| Joshua Hursey | Yutaka Ishikawa | Bin Jia |
| Hideyuki Jitsumoto | Terry Jones | Yann Kalemkarian |
| Ranier Keller | Matthew Koop | Quincey Koziol |
| Manojkumar Krishnan | Sameer Kumar | Miron Livny |
| Andrew Lumsdaine | Miao Luo | Ewing Lusk |
| Timothy I. Mattox | Kannan Narasimhan | Mark Pagel |
| Avneesh Pant | Steve Poole | Howard Pritchard |
| Craig Rasmussen | Hubert Ritzdorf | Rob Ross |
| Martin Schulz | Pavel Shamis | Galen Shipman |
| Christian Siebert | Anthony Skjellum | Brian Smith |
| Naoki Sueyasu | Vinod Tipparaju | Keith Underwood |
| Rolf Vandevaart | Abhinav Vishnu | Weikuan Yu |

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

The following institutions supported the MPI-2.2 effort through time and travel support for the people listed above.

Argonne National Laboratory
Auburn University
Bull
Cisco Systems, Inc.
Cray Inc.
Forschungszentrum Jülich
Fujitsu
The HDF Group
Hewlett-Packard
International Business Machines
Indiana University
Institut National de Recherche en Informatique et Automatique (INRIA)
Institute for Advanced Science & Engineering Corporation
Intel Corporation
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
Mathworks
Mellanox Technologies
Microsoft
Myricom
NEC Corporation

Oak Ridge National Laboratory
The Ohio State University
Pacific Northwest National Laboratory
QLogic Corporation
RunTime Computing Solutions, LLC
Sandia National Laboratories
SiCortex, Inc.
Silicon Graphics Inc.
Sun Microsystems, Inc.
Tokyo Institute of Technology
University of Alabama at Birmingham
University of Houston
University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Tokyo
University of Wisconsin

Funding for the MPI Forum meetings was partially supported by awards #CCF-0816909 and #CCF-1144042 from the National Science Foundation. In addition, the HDF Group provided travel support for one U.S. academic.

## MPI-3.0:

MPI-3.0 is a signficant effort to extend and modernize the MPI Standard.
The editors and organizers of the MPI-3.0 have been:

- William Gropp, Steering committee, Front matter, Introduction, Groups, Contexts, and Communicators, One-Sided Communications, and Bibliography
- Richard Graham, Steering committee, Point-to-Point Communication, Meeting Convener, and MPI-3.0 chair
- Torsten Hoefler, Collective Communication, One-Sided Communications, and Process Topologies
- George Bosilca, Datatypes and Environmental Management
- David Solt, Process Creation and Management
- Bronis R. de Supinski, External Interfaces and Tool Support
- Rajeev Thakur, I/O and One-Sided Communications
- Darius Buntinas, Info Object
- Jeffrey M. Squyres, Language Bindings and MPI-3.0 Secretary
- Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings
- Craig Rasmussen, Fortran Bindings

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

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

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

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

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

Argonne National Laboratory
Bull
Cisco Systems, Inc.
Cray Inc.
CSCS

ETH Zurich
Fujitsu Ltd.
German Research School for Simulation Sciences
The HDF Group
Hewlett-Packard
International Business Machines
IBM India Private Ltd
Indiana University
Institut National de Recherche en Informatique et Automatique (INRIA)
Institute for Advanced Science & Engineering Corporation
Intel Corporation
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Los Alamos National Laboratory
Mellanox Technologies, Inc.
Microsoft Corporation
NEC Corporation
National Oceanic and Atmospheric Administration, Global Systems Division
NVIDIA Corporation
Oak Ridge National Laboratory
The Ohio State University
Oracle America
Platform Computing
RIKEN AICS
RunTime Computing Solutions, LLC
Sandia National Laboratories
Technical University of Chemnitz
Tokyo Institute of Technology
University of Alabama at Birmingham
University of Chicago
University of Houston
University of Illinois at Urbana-Champaign
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Tokyo

Funding for the MPI Forum meetings was partially supported by awards #CCF-0816909 and #CCF-1144042 from the National Science Foundation. In addition, the HDF Group and Sandia National Laboratories provided travel support for one U.S. academic each.

## MPI-3.1:

MPI-3.1 is a minor update to the MPI Standard.
The editors and organizers of the MPI-3.1 have been:

- Martin Schulz, MPI-3.1 chair
- William Gropp, Steering committee, Front matter, Introduction, One-Sided Communications, and Bibliography; Overall editor

- Rolf Rabenseifner, Steering committee, Terms and Definitions, and Fortran Bindings, Deprecated Functions, Annex Change-Log, and Annex Language Bindings
- Richard L. Graham, Steering committee, Meeting Convener
- Jeffrey M. Squyres, Language Bindings and MPI-3.1 Secretary
- Daniel Holmes, Point-to-Point Communication
- George Bosilca, Datatypes and Environmental Management
- Torsten Hoefler, Collective Communication and Process Topologies
- Pavan Balaji, Groups, Contexts, and Communicators, and External Interfaces
- Jeff Hammond, The Info Object
- David Solt, Process Creation and Management
- Quincey Koziol, I/O
- Kathryn Mohror, Tool Support
- Rajeev Thakur, One-Sided Communications

The following list includes some of the active participants who attended MPI Forum meetings or participated in the e-mail discussions.

| | | |
|---|---|---|
| Charles Archer | Pavan Balaji | Purushotham V. Bangalore |
| Brian Barrett | Wesley Bland | Michael Blocksome |
| George Bosilca | Aurelien Bouteiller | Devendar Bureddy |
| Yohann Burette | Mohamad Chaarawi | Alexey Cheptsov |
| James Dinan | Dmitry Durnov | Thomas Francois |
| Edgar Gabriel | Todd Gamblin | Balazs Gerofi |
| Paddy Gillies | David Goodell | Manjunath Gorentla Venkata |
| Richard L. Graham | Ryan E. Grant | William Gropp |
| Khaled Hamidouche | Jeff Hammond | Amin Hassani |
| Marc-André Hermanns | Nathan Hjelm | Torsten Hoefler |
| Daniel Holmes | Atsushi Hori | Yutaka Ishikawa |
| Hideyuki Jitsumoto | Jithin Jose | Krishna Kandalla |
| Christos Kavouklis | Takahiro Kawashima | Chulho Kim |
| Michael Knobloch | Alice Koniges | Quincey Koziol |
| Sameer Kumar | Joshua Ladd | Ignacio Laguna |
| Huiwei Lu | Guillaume Mercier | Kathryn Mohror |
| Adam Moody | Tomotake Nakamura | Takeshi Nanri |
| Steve Oyanagi | Antonio J. Pẽna | Sreeram Potluri |
| Howard Pritchard | Rolf Rabenseifner | Nicholas Radcliffe |
| Ken Raffenetti | Raghunath Raja | Craig Rasmussen |
| Davide Rossetti | Kento Sato | Martin Schulz |
| Sangmin Seo | Christian Siebert | Anthony Skjellum |
| Brian Smith | David Solt | Jeffrey M. Squyres |

| | | |
|---|---|---|
| Hari Subramoni | Shinji Sumimoto | Alexander Supalov |
| Bronis R. de Supinski | Sayantan Sur | Masamichi Takagi |
| Keita Teranishi | Rajeev Thakur | Fabian Tillier |
| Yuichi Tsujita | Geoffroy Vallée | Rolf vandeVaart |
| Akshay Venkatesh | Jerome Vienne | Venkat Vishwanath |
| Anh Vo | Huseyin S. Yildiz | Junchao Zhang |
| Xin Zhao | | |

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

The following institutions supported the MPI-3.1 effort through time and travel support for the people listed above.

Argonne National Laboratory
Auburn University
Cisco Systems, Inc.
Cray
EPCC, The University of Edinburgh
ETH Zurich
Forschungszentrum Jülich
Fujitsu
German Research School for Simulation Sciences
The HDF Group
International Business Machines
INRIA
Intel Corporation
Jülich Aachen Research Alliance, High-Performance Computing (JARA-HPC)
Kyushu University
Lawrence Berkeley National Laboratory
Lawrence Livermore National Laboratory
Lenovo
Los Alamos National Laboratory
Mellanox Technologies, Inc.
Microsoft Corporation
NEC Corporation
NVIDIA Corporation
Oak Ridge National Laboratory
The Ohio State University
RIKEN AICS
Sandia National Laboratories
Texas Advanced Computing Center
Tokyo Institute of Technology
University of Alabama at Birmingham
University of Houston
University of Illinois at Urbana-Champaign
University of Oregon
University of Stuttgart, High Performance Computing Center Stuttgart (HLRS)
University of Tennessee, Knoxville
University of Tokyo

MPI-4.0:

MPI-4.0 is a major update to the MPI Standard.

# Chapter 1

# Introduction to MPI

## 1.1 Overview and Goals

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

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 for which they can provide hardware support, thereby enhancing scalability.

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

A complete list of goals follows.

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

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

## 1.2 Background of MPI-1.0

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 [50], Express [13], nCUBE's Vertex [46], p4 [8, 9], and PARMACS [5, 10]. Other important contributions have come from Zipcode [52, 53], Chimp [19, 20], PVM [4, 17], Chameleon [27], and PICL [25].

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 [59]. 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 MPI-1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [18]. MPI-1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI-1 was primarily intended to promote discussion and "get the ball rolling," it focused mainly on point-to-point communications. MPI-1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at 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 [22]. The first product of these deliberations was Version 1.1 of the MPI specification, released in June of 1995 [23] (see http://www.mpi-forum.org for official MPI document releases). At that time, effort focused in five areas.

1. Further corrections and clarifications for the MPI-1.1 document.

2. Additions to MPI-1.1 that do not significantly change its types of functionality (new datatype constructors, language interoperability, etc.).

3. Completely new types of functionality (dynamic processes, one-sided communication, parallel I/O, etc.) that are what everyone thinks of as "MPI-2 functionality."

4. Bindings for Fortran 90 and C++. MPI-2 specifies C++ bindings for both MPI-1 and MPI-2 functions, and extensions to the Fortran 77 binding of MPI-1 and MPI-2 to handle Fortran 90 issues.

5. Discussions of areas in which the MPI process and framework seem likely to be useful, but where more discussion and experience are needed before standardization (e.g., zero-copy semantics on shared-memory machines, real-time specifications).

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

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

- MPI-1 compliance will mean compliance with MPI-1.3. This is a useful level of compliance. It means that the implementation conforms to the clarifications of MPI-1.1 function behavior given in Chapter 3 of the MPI-2 document. Some implementations may require changes to be MPI-1 compliant.

- MPI-2 compliance will mean compliance with all of MPI-2.1.

- The MPI Journal of Development is not part of the MPI Standard.

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

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

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 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 Background of MPI-2.2

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

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

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

## 1.6 Background of MPI-3.0

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

## 1.7 Background of MPI-3.1

MPI-3.1 is a minor update to the MPI standard. Most of the updates are corrections and clarifications to the standard, especially for the Fortran bindings. New functions added include routines to manipulate `MPI_Aint` values in a portable manner, nonblocking collective I/O routines, and routines to get the index value by name for MPI_T performance and control variables. A general index was also added.

## 1.8 Background of MPI-4.0

## 1.9 Who Should Use This Standard?

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

## 1.10 What Platforms Are Targets for Implementation?

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations, including those for multi-core processors and hybrid architectures, are possible. The paradigm will not be made obsolete by architectures combining the 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 heterogenous networks of workstations.

## 1.11 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 and C,
- Tool support.

## 1.12 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.

## 1.13 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 Communication, defines process-group collective communication operations. Well known examples of this are barrier and broadcast over a group of processes (not necessarily all the processes). With MPI-2, the semantics of collective communication was extended to include intercommunicators. It also adds two new collective operations. MPI-3 adds nonblocking 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 in 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 **??**, **??**, covers interfaces that allow debuggers, performance analyzers, and other tools to obtain data about the operation of MPI processes. This chapter includes Section **??** (**??**), which was a chapter in previous versions of MPI.

- Chapter 14, Deprecated Interfaces, describes routines that are kept for reference. However usage of these functions is discouraged, as they may be deleted in future versions of the standard.

- Chapter 15, Removed Interfaces, describes routines and constructs that have been removed from MPI. Some of these were deprecated in MPI-2, and the MPI Forum decided to remove these from the MPI-3 standard. Others of these were deprecated in MPI-3, and the MPI Forum decided to remove these from the MPI-4 standard.

- Chapter 16, Backward Incompatibilities, describes incompatibilities with previous versions of MPI.

- Chapter 17, Language Bindings, discusses Fortran issues, and describes language interoperability aspects between C and Fortran.

The Appendices are:

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

- Annex B, Change-Log, summarizes some changes since the previous version of the standard.
- Several Index pages show the locations of examples, constants and predefined handles, callback routine 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 during the MPI-2 development and deemed to have value, but are not included in the MPI Standard. They are part of the "Journal of Development" (JOD), lest good ideas be lost and in order to provide a starting point for further work. The chapters in the JOD are

- 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.

## 2.1 Document Notation

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

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

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

## 2.2 Naming Conventions

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

1. In C, all routines associated with a particular type of MPI object should be of the form MPI_Class_action_subset or, if no subset exists, of the form MPI_Class_action. In Fortran, all routines associated with a particular type of MPI object should be of the form MPI_CLASS_ACTION_SUBSET or, if no subset exists, of the form MPI_CLASS_ACTION.

2. If the routine is not associated with a class, the name should be of the form MPI_Action_subset in C and MPI_ACTION_SUBSET in Fortran.

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

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

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 from the perspective of the caller at any time during the call's execution,
- OUT: the call may update the argument but does not use its input value,
- INOUT: the call may both use and update the argument.

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

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

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

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

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

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

```
void copyIntBuffer(int *pin, int *pout, int len)
{   int i;
    for (i=0; i<len; ++i) *pout++ = *pin++;
}
```

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

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

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

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

- The ISO C version of the function.
- The Fortran version used with `USE mpi_f08`.
- The Fortran version of the same function used with `USE mpi` or `INCLUDE 'mpif.h'`.

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

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

## 2.4 Semantic Terms

When discussing MPI procedures the following semantic terms are used.

**nonblocking** A procedure is nonblocking if it may return before the associated operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. The word complete is used with respect to operations and any associated requests and/or communications. An **operation completes** when the user is allowed to reuse resources, and any output buffers have been updated.

**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_INT, or MPI_PACKED) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or MPI_TYPE_CREATE_F90_COMPLEX. The former are **named** whereas the latter are **unnamed**.

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

**portable** A datatype is portable if it is a predefined datatype, or it is derived from 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_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, 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 with `USE mpi` or `INCLUDE 'mpif.h'`, all handles have type `INTEGER`. In Fortran with `USE mpi_f08`, and in C, a different handle type is defined for each category of objects. With Fortran `USE mpi_f08`, the handles are defined as Fortran `BIND(C)` derived types that consist of only one element `INTEGER :: MPI_VAL`. The internal handle value is identical to the Fortran `INTEGER` value used in the `mpi` module and `mpif.h`. The operators `.EQ.`, `.NE.`, `==` and `/=` are overloaded to allow the comparison of these handles. The type names are identical to the names in C, except that they are not case sensitive. For example:

```
TYPE, BIND(C) :: MPI_Comm
  INTEGER   :: MPI_VAL
END TYPE MPI_Comm
```

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

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

*Rationale.* Since the Fortran integer values are equivalent, applications can easily convert MPI handles between all three supported Fortran methods. For example, an integer communicator handle `COMM` can be converted directly into an exactly equivalent `mpi_f08` communicator handle named `comm_f08` by `comm_f08%MPI_VAL=COMM`, and vice versa. The use of the `INTEGER` defined handles and the `BIND(C)` derived type handles is different: Fortran 2003 (and later) define that `BIND(C)` derived types can be used within user defined common blocks, but it is up to the rules of the companion C compiler how many numerical storage units are used for these `BIND(C)` derived type handles. Most compilers use one unit for both, the `INTEGER` handles and the handles defined as `BIND(C)` derived types. (*End of rationale.*)

*Advice to users.* If a user wants to substitute `mpif.h` or the `mpi` module by the `mpi_f08` module and the application program stores a handle in a Fortran common block then it is necessary to change the Fortran support method in all application routines that use this common block, because the number of numerical storage units of such a handle can be different in the two modules. (*End of advice to users.*)

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.

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

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

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

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

*Advice to implementors.* The intended semantics of opaque objects is that opaque 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 then 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, see Chapter 8. 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).

MPI also provides predefined named constant handles, such as MPI_COMM_WORLD.

All named constants, with the exceptions noted below for Fortran, can be used in initialization expressions or assignments, but not necessarily in array declarations or as labels in C `switch` or Fortran `select/case` statements. This implies named constants to be link-time but not necessarily compile-time constants. The named constants listed below are required to be compile-time constants in both C and Fortran. These constants do not change values during execution. Opaque objects accessed by constant handles are defined and do not change value between MPI initialization (MPI_INIT) and MPI completion (MPI_FINALIZE). The handles themselves are constants and can be also used in initialization expressions or assignments.

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

MPI_MAX_PROCESSOR_NAME
MPI_MAX_LIBRARY_VERSION_STRING
MPI_MAX_ERROR_STRING
MPI_MAX_DATAREP_STRING
MPI_MAX_INFO_KEY
MPI_MAX_INFO_VAL
MPI_MAX_OBJECT_NAME
MPI_MAX_PORT_NAME
MPI_VERSION
MPI_SUBVERSION
MPI_STATUS_SIZE (Fortran only)
MPI_ADDRESS_KIND (Fortran only)
MPI_COUNT_KIND (Fortran only)
MPI_INTEGER_KIND (Fortran only)
MPI_OFFSET_KIND (Fortran only)
MPI_SUBARRAYS_SUPPORTED (Fortran only)
MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only)

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

MPI_BOTTOM
MPI_STATUS_IGNORE
MPI_STATUSES_IGNORE

MPI_ERRCODES_IGNORE
MPI_IN_PLACE
MPI_ARGV_NULL
MPI_ARGVS_NULL
MPI_UNWEIGHTED
MPI_WEIGHTS_EMPTY

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

### 2.5.5 Choice

MPI functions sometimes use arguments with a *choice* (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types. The mechanism for providing such arguments will differ from language to language. For Fortran with the include file `mpif.h` or the `mpi` module, the document uses <type> to represent a choice variable; with the Fortran `mpi_f08` module, such arguments are declared with the Fortran 2008 + TR 29113 syntax `TYPE(*), DIMENSION(..)`; for C, we use `void *`.

*Advice to implementors.* Implementors can freely choose how to implement choice arguments in the `mpi` module, e.g., with a non-standard compiler-dependent method that has the quality of the call mechanism in the implicit Fortran interfaces, or with the method defined for the `mpi_f08` module. See details in Section 17.1.1. (*End of advice to implementors.*)

### 2.5.6 Absolute Addresses and Relative Address Displacements

Some MPI procedures use *address* arguments that represent an *absolute address* in the calling program, or *relative displacement* arguments that represent differences of two absolute addresses. The datatype of such arguments is MPI_Aint in C and `INTEGER (KIND=MPI_ADDRESS_KIND)` in Fortran. These types must have the same width and encode address values in the same manner such that address values in one language may be passed directly to another language without conversion. There is the MPI constant MPI_BOTTOM to indicate the start of the address range. For retrieving absolute addresses or any calculation with absolute addresses, one should use the routines and functions provided in Section 4.1.5. Section 4.1.12 provides additional rules for the correct use of absolute addresses. For expressions with relative displacements or other usage without absolute addresses, intrinsic operators (e.g., `+`, `-`, `*`) can be used.

### 2.5.7 File Offsets

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

### 2.5.8 Counts

As described above, MPI defines types (e.g., MPI_Aint) to address locations within memory and other types (e.g., MPI_Offset) to address locations within files. In addition, some MPI procedures use *count* arguments that represent a number of MPI datatypes on which to operate. At times, one needs a single type that can be used to address locations within either memory or files as well as express *count* values, and that type is MPI_Count in C and INTEGER (KIND=MPI_COUNT_KIND) in Fortran. These types must have the same width and encode values in the same manner such that count values in one language may be passed directly to another language without conversion. The size of the MPI_Count type is determined by the MPI implementation with the restriction that it must be minimally capable of encoding any value that may be stored in a variable of type int, MPI_Aint, or MPI_Offset in C and of type INTEGER, INTEGER (KIND=MPI_ADDRESS_KIND), or INTEGER (KIND=MPI_OFFSET_KIND) in Fortran.

*Rationale.* Count values logically need to be large enough to encode any value used for expressing element counts, type maps in memory, type maps in file views, etc. For backward compatibility reasons, many MPI routines still use int in C and INTEGER in Fortran as the type of count arguments. (*End of rationale.*)

## 2.6 Language Binding

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

MPI bindings are for Fortran 90 or later, though they were originally designed to be usable in Fortran 77 environments. With the mpi_f08 module, two new Fortran features, *assumed type* and *assumed rank*, are also required, see Section 2.5.5.

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

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

### 2.6.1 Deprecated and Removed Interfaces

A number of chapters refer to deprecated or replaced MPI constructs. These are constructs that continue to be part of the MPI standard, as documented in Chapter 14, but that users are recommended not to continue using, since better solutions were provided with newer versions of MPI. 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 was declared as 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.

Some of the deprecated constructs are now removed, as documented in Chapter 15. They may still be provided by an implementation for backwards compatibility, but are not required.

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

| Deprecated or removed construct | deprecated since | removed since | Replacement |
|---|---|---|---|
| MPI_ADDRESS | MPI-2.0 | MPI-3.0 | MPI_GET_ADDRESS |
| MPI_TYPE_HINDEXED | MPI-2.0 | MPI-3.0 | MPI_TYPE_CREATE_HINDEXED |
| MPI_TYPE_HVECTOR | MPI-2.0 | MPI-3.0 | MPI_TYPE_CREATE_HVECTOR |
| MPI_TYPE_STRUCT | MPI-2.0 | MPI-3.0 | MPI_TYPE_CREATE_STRUCT |
| MPI_TYPE_EXTENT | MPI-2.0 | MPI-3.0 | MPI_TYPE_GET_EXTENT |
| MPI_TYPE_UB | MPI-2.0 | MPI-3.0 | MPI_TYPE_GET_EXTENT |
| MPI_TYPE_LB | MPI-2.0 | MPI-3.0 | MPI_TYPE_GET_EXTENT |
| MPI_LB[1] | MPI-2.0 | MPI-3.0 | MPI_TYPE_CREATE_RESIZED |
| MPI_UB[1] | MPI-2.0 | MPI-3.0 | MPI_TYPE_CREATE_RESIZED |
| MPI_ERRHANDLER_CREATE | MPI-2.0 | MPI-3.0 | MPI_COMM_CREATE_ERRHANDLER |
| MPI_ERRHANDLER_GET | MPI-2.0 | MPI-3.0 | MPI_COMM_GET_ERRHANDLER |
| MPI_ERRHANDLER_SET | MPI-2.0 | MPI-3.0 | MPI_COMM_SET_ERRHANDLER |
| MPI_Handler_function[2] | MPI-2.0 | MPI-3.0 | MPI_Comm_errhandler_function[2] |
| MPI_KEYVAL_CREATE | MPI-2.0 | | MPI_COMM_CREATE_KEYVAL |
| MPI_KEYVAL_FREE | MPI-2.0 | | MPI_COMM_FREE_KEYVAL |
| MPI_DUP_FN[3] | MPI-2.0 | | MPI_COMM_DUP_FN[3] |
| MPI_NULL_COPY_FN[3] | MPI-2.0 | | MPI_COMM_NULL_COPY_FN[3] |
| MPI_NULL_DELETE_FN[3] | MPI-2.0 | | MPI_COMM_NULL_DELETE_FN[3] |
| MPI_Copy_function[2] | MPI-2.0 | | MPI_Comm_copy_attr_function[2] |
| COPY_FUNCTION[3] | MPI-2.0 | | COMM_COPY_ATTR_FUNCTION[3] |
| MPI_Delete_function[2] | MPI-2.0 | | MPI_Comm_delete_attr_function[2] |
| DELETE_FUNCTION[3] | MPI-2.0 | | COMM_DELETE_ATTR_FUNCTION[3] |
| MPI_ATTR_DELETE | MPI-2.0 | | MPI_COMM_DELETE_ATTR |
| MPI_ATTR_GET | MPI-2.0 | | MPI_COMM_GET_ATTR |
| MPI_ATTR_PUT | MPI-2.0 | | MPI_COMM_SET_ATTR |
| MPI_COMBINER_HVECTOR_INTEGER[4] | - | MPI-3.0 | MPI_COMBINER_HVECTOR[4] |
| MPI_COMBINER_HINDEXED_INTEGER[4] | - | MPI-3.0 | MPI_COMBINER_HINDEXED[4] |
| MPI_COMBINER_STRUCT_INTEGER[4] | - | MPI-3.0 | MPI_COMBINER_STRUCT[4] |
| MPI::... | MPI-2.2 | MPI-3.0 | C language binding |
| MPI_CANCEL for send requests | MPI-3.2 | | no direct replacement |

[1] Predefined datatype.
[2] Callback prototype definition.
[3] Predefined callback routine.
[4] Constant.
Other entries are regular MPI routines.

Table 2.1: Deprecated and Removed constructs

### 2.6.2 Fortran Binding Issues

Originally, MPI-1.1 provided bindings for Fortran 77. These bindings are retained, but they are now interpreted in the context of the Fortran 90 standard. MPI can still be used with most Fortran 77 compilers, as noted below. When the term “Fortran” is used it means Fortran 90 or later; it means Fortran 2008 + TR 29113 and later if the mpi_f08 module is used.

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

All MPI Fortran subroutines have a return code in the last argument. With USE mpi_f08, this last argument is declared as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g., MPI_NULL_COPY_FN). A few MPI operations which are functions do not have the return code argument. The return code value for successful completion is MPI_SUCCESS. Other error codes are implementation dependent; see the error codes in Chapter 8 and Annex A.

Constants representing the maximum length of a string are one smaller in Fortran than in C as discussed in Section 17.2.9.

Handles are represented in Fortran as INTEGERs, or as a BIND(C) derived type with the mpi_f08 module; see Section 2.5.1. Binary-valued variables are of type LOGICAL.

Array arguments are indexed from one.

The older MPI Fortran bindings (mpif.h and use mpi) are inconsistent with the Fortran standard in several respects. These inconsistencies, such as register optimization problems, have implications for user codes that are discussed in detail in Section 17.1.16.

### 2.6.3 C Binding Issues

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

The definition of named constants, function prototypes, and type definitions must be supplied in an include file mpi.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 *.

### 2.6.4 Functions and Macros

An implementation is allowed to implement MPI_WTIME, PMPI_WTIME, MPI_WTICK, PMPI_WTICK, MPI_AINT_ADD, PMPI_AINT_ADD, MPI_AINT_DIFF, PMPI_AINT_DIFF, and the handle-conversion functions (MPI_Group_f2c, etc.) in Section 17.2.4, 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.*)

## 2.7 Processes

An MPI program consists of autonomous processes, executing their own code, in an 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.

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. On the other hand, some errors may be detected after the associated operation has completed; some errors may not have a communicator, window, or file on which an error may be raised. In such cases, these errors will be raised on the communicator MPI_COMM_SELF.

An example of such a case 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).

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).

```
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 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).

```
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.

```
#include "mpi.h"
int main(int argc, char *argv[])
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  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();
  return 0;
}
```

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, the send operation associates an **envelope** with the message. This envelope specifies the message destination and contains distinguishing information that can be used by the **receive** operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the envelope for the message sent. Process one (`myrank = 1`) receives this message with the **receive** operation MPI_RECV. The message to be received is selected according to the value of its envelope, and the message data is stored into the **receive buffer**. In the example above, the receive buffer consists of the storage containing the string message in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.

The next sections describe the blocking send and receive operations. We discuss send, receive, blocking communication semantics, type matching requirements, type conversion in heterogeneous environments, and more general communication modes. Nonblocking communication is addressed next, followed by probing and canceling a message, channel-like constructs and send-receive operations, ending with a description of the "dummy" process, MPI_PROC_NULL.

## 3.2 Blocking Send and Receive Operations

### 3.2.1 Blocking Send

The syntax of the blocking send operation is given below.

MPI_SEND(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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |

**C binding**

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
```

**F08 binding**

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

**F binding**

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```

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

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

### 3.2.2 Message Data

The send buffer specified by the MPI_SEND 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.

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

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

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

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

MPI requires support of these datatypes, which match the basic datatypes of Fortran and ISO C. Additional MPI datatypes should be provided if the host language has additional data types: MPI_DOUBLE_COMPLEX for double precision complex in Fortran declared to be of type `DOUBLE COMPLEX`; MPI_REAL2, MPI_REAL4, and MPI_REAL8 for Fortran reals, declared to be of type `REAL*2`, `REAL*4` and `REAL*8`, respectively; MPI_INTEGER1, MPI_INTEGER2, and MPI_INTEGER4 for Fortran integers, declared to be of type `INTEGER*1`, `INTEGER*2`, and `INTEGER*4`, respectively; etc.

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

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | char<br>(treated as printable character) |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_LONG_LONG_INT | signed long long int |
| MPI_LONG_LONG (as a synonym) | signed long long int |
| MPI_SIGNED_CHAR | signed char<br>(treated as integral value) |
| MPI_UNSIGNED_CHAR | unsigned char<br>(treated as integral value) |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_UNSIGNED_LONG_LONG | unsigned long long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_WCHAR | wchar_t<br>(defined in <stddef.h>)<br>(treated as printable character) |
| MPI_C_BOOL | _Bool |
| MPI_INT8_T | int8_t |
| MPI_INT16_T | int16_t |
| MPI_INT32_T | int32_t |
| MPI_INT64_T | int64_t |
| MPI_UINT8_T | uint8_t |
| MPI_UINT16_T | uint16_t |
| MPI_UINT32_T | uint32_t |
| MPI_UINT64_T | uint64_t |
| MPI_C_COMPLEX | float _Complex |
| MPI_C_FLOAT_COMPLEX (as a synonym) | float _Complex |
| MPI_C_DOUBLE_COMPLEX | double _Complex |
| MPI_C_LONG_DOUBLE_COMPLEX | long double _Complex |
| MPI_BYTE | |
| MPI_PACKED | |

Table 3.2: Predefined MPI datatypes corresponding to C datatypes

The need for such datatype information will become clear in Section 3.3.2. (*End of rationale.*)

The datatypes MPI_AINT, MPI_OFFSET, and MPI_COUNT correspond to the MPI-defined C types MPI_Aint, MPI_Offset, and MPI_Count and their Fortran equivalents INTEGER (KIND=MPI_ADDRESS_KIND), INTEGER (KIND=MPI_OFFSET_KIND), and INTEGER (KIND=MPI_COUNT_KIND). This is described in Table 3.3. All predefined datatype handles

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

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

are available in all language bindings. See Sections 17.2.6 and 17.2.10 on page 674 and 682 for information on interlanguage communication with these types.

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

| MPI datatype | C++ datatype |
|---|---|
| MPI_CXX_BOOL | bool |
| MPI_CXX_FLOAT_COMPLEX | std::complex<float> |
| MPI_CXX_DOUBLE_COMPLEX | std::complex<double> |
| MPI_CXX_LONG_DOUBLE_COMPLEX | std::complex<long double> |

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

### 3.2.3 Message Envelope

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

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 $0, \ldots,$ 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.

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 $0, \ldots, n-1 \cup \{$MPI_PROC_NULL$\}$, 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.

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 or MPI_ANY_SOURCE (non-negative integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | status | status object (Status) |

**C binding**

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

**F08 binding**

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
```

```
    INTEGER, INTENT(IN) ::  count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

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 $\{0, \ldots, n-1\} \cup$ {MPI_ANY_SOURCE} $\cup$ {MPI_PROC_NULL}, where $n$ is the number of processes in this group.

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

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

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

The use of dest or source=MPI_PROC_NULL to define a "dummy" destination or source in any send or receive call is described in Section 3.11.

### 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 with `USE mpi` or `INCLUDE 'mpif.h'`, status is an array of `INTEGER`s of size MPI_STATUS_SIZE. The constants MPI_SOURCE, MPI_TAG and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_SOURCE), status(MPI_TAG) and status(MPI_ERROR) contain, respectively, the source, tag and error code of the received message.

With Fortran `USE mpi_f08`, status is defined as the Fortran `BIND(C)` derived type `TYPE(MPI_Status)` containing three public `INTEGER` fields named MPI_SOURCE, MPI_TAG, and MPI_ERROR. `TYPE(MPI_Status)` may contain additional, implementation-specific fields. Thus, `status%MPI_SOURCE`, `status%MPI_TAG` and `status%MPI_ERROR` contain the source, tag, and error code of a received message respectively. Additionally, within both the `mpi` and the `mpi_f08` modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and `TYPE(MPI_Status)` are defined to allow conversion between both status representations. Conversion routines are provided in Section 17.2.5.

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

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

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

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

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

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 (non-negative integer) |

**C binding**

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

**F08 binding**

```
MPI_Get_count(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

Returns the number of entries received. (Again, we count *entries*, each of type *datatype*, not *bytes*.) The datatype argument should match the argument provided by the receive call that set the status variable. If the number of entries received exceeds the limits of the count parameter, then MPI_GET_COUNT sets the value of count to MPI_UNDEFINED. There are other situations where the value of count can be set to MPI_UNDEFINED; see Section 4.1.11.

> *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, probe, wait, or test function, inform the implementation that the status fields are not to be filled in. Note that MPI_STATUS_IGNORE is not a special type of MPI_Status object; rather, it is a special value for the argument. In C one would expect it to be NULL, not the address of a special MPI_Status.

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

In general, this optimization can apply to all functions for which status or an array of statuses is an OUT argument. Note that this converts status into an INOUT argument. The functions that can be passed MPI_STATUS_IGNORE are all the various forms of MPI_RECV, MPI_PROBE, MPI_TEST, and MPI_WAIT, as well as MPI_REQUEST_GET_STATUS. When an array is passed, as in the MPI_{TEST|WAIT}{ALL|SOME} functions, a separate constant, MPI_STATUSES_IGNORE, is passed for the array argument. It is possible for an MPI function to return MPI_ERR_IN_STATUS even when MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE has been passed to that function.

MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE are not required to have the same values in C and Fortran.

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.

## 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.

The following examples illustrate the first two cases.

**Example 3.1** Sender and receiver specify matching types.

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

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

**Example 3.2** Sender and receiver do not specify matching types.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(b(1), 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 3.3** Sender and receiver specify communication of untyped values.

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

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

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

### Type MPI_CHARACTER

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

**Example 3.4**

Transfer of Fortran CHARACTERs.

```
CHARACTER*10 a
CHARACTER*10 b

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

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

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

A Fortran character variable is a constant length string, with no special termination symbol. There is no fixed convention on how to represent characters, and how to store their length. Some compilers pass a character argument to a routine as a pair of arguments, one holding the address of the string and the other holding the length of string. Consider the case of an MPI communication call that is passed a communication buffer with type defined by a derived datatype (Section 4.1). If this communicator buffer contains variables of type CHARACTER then the information on their length will not be passed to the MPI routine.

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

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

### 3.3.2 Data Conversion

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

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

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

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

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

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

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

Consider the three examples, 3.1–3.3. 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 implementors.* The current definition does not require messages to carry data type information. Both sender and receiver provide complete data type information. In a heterogeneous environment, one can either use a machine independent encoding such as XDR, or have the receiver convert from the sender representation to its own, or even have the sender do the conversion.
>
> Additional type information might be added to messages in order to allow the system to detect mismatches between datatype at sender and receiver. This might be particularly useful in a slower but safer debug mode. (*End of advice to implementors.*)

MPI requires support for inter-language communication, 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 17.2.

## 3.4 Communication Modes

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

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

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

Thus, a 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.

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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |

**C binding**

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

**F08 binding**

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

**F binding**

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |

**C binding**

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

**F08 binding**

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

```
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |

**C binding**

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

**F08 binding**

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

**F binding**

```
MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

Send in ready mode.

There is only one receive operation, but it matches any of the send modes. The receive 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).

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

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

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

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

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

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

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

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

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

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

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

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

## 3.5 Semantics of Point-to-Point Communication

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

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

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 multithreaded, 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 3.5** An example of non-overtaking messages.

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

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

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 3.6** An example of two, intertwined matching pairs.

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

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

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

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

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

A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signaled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values 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 3.7** An exchange of messages.

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

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

**Example 3.8** 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)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

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

**Example 3.9** 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)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
ELSE IF (rank.EQ.1) THEN
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

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

> *Advice to users.* When standard send operations are used, then a deadlock situation may occur where both processes are blocked because buffer space is not available. The same will certainly happen, if the synchronous mode is used. If the buffered mode is used, and not enough buffer space is available, then the program will not complete either. However, rather than a deadlock situation, we shall have a buffer overflow error.
>
> A program is "safe" if no message buffering is required for the program to complete. One can replace all sends in such program with synchronous sends, and the program will still run correctly. This conservative programming style provides the best portability, since program completion does not depend on the amount of buffer space available or on the communication protocol used.
>
> Many programmers prefer to have more leeway and opt to use the "unsafe" programming style shown in Example 3.9. In such cases, the use of standard sends is likely to provide the best compromise between performance and robustness: quality implementations will provide sufficient buffering so that "common practice" programs will not deadlock. The buffered send mode can be used for programs that require more buffering, or in situations where the programmer wants more control. This mode

might also be used for debugging purposes, as buffer overflow conditions are easier to diagnose than deadlock conditions.

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

## 3.6 Buffer Allocation and Usage

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

MPI_BUFFER_ATTACH(buffer, size)

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

**C binding**

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

**F08 binding**

```
MPI_Buffer_attach(buffer, size, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buffer
    INTEGER, INTENT(IN) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

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. In C, buffer is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 17.1.12).

MPI_BUFFER_DETACH(buffer_addr, size)

| | | |
|---|---|---|
| OUT | buffer_addr | initial buffer address |
| OUT | size | buffer size, in bytes (non-negative integer) |

**C binding**

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

**F08 binding**

```
MPI_Buffer_detach(buffer_addr, size, ierror)
```

```
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) BUFFER_ADDR
    INTEGER 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 3.10** Calls to attach and detach buffers.

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

*Advice to users.* Even though the C functions MPI_Buffer_attach and MPI_Buffer_detach both have a first argument of type void*, these arguments are used differently: A pointer to the buffer is passed to MPI_Buffer_attach; the address of the pointer is passed to MPI_Buffer_detach, so that this call can return the pointer value. In Fortran with the `mpi` module or `mpif.h`, the type of the buffer_addr argument is wrongly defined and the argument is therefore unused. In Fortran with the `mpi_f08` module, the address of the buffer is returned as `TYPE(C_PTR)`, see also Example 8.1 about the use of `C_PTR` pointers. (*End of advice to users.*)

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

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

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

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

> *Rationale.* There is a wide spectrum of possible implementations of buffered communication: buffering can be done at sender, at receiver, or both; buffers can be 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 MPI_PACK_SIZE(count, datatype, comm, size), with the count, datatype and comm arguments used in the MPI_BSEND call, returns an upper bound on the amount of space needed to buffer the message data (see Section 4.2). The MPI constant MPI_BSEND_OVERHEAD provides an upper bound on the additional space consumed by the entry (e.g., for pointers or envelope information).
- Find the next contiguous empty space of $n$ bytes in buffer (space following queue tail, or space at start of buffer if queue tail is too close to end of buffer). If space is not found then raise buffer overflow error.
- Append to end of PME queue in contiguous space the new entry that contains request handle, next pointer and packed message data; MPI_PACK is used to pack data.
- Post nonblocking send (standard mode) for packed data.
- Return

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

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

### 3.7.1 Communication Request Objects

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

### 3.7.2 Communication Initiation

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)

| | | |
|---|---|---|
| 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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

Start a standard mode, nonblocking send.

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

| | | |
|---|---|---|
| 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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

Start a buffered mode, nonblocking send.

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

| | | |
|---|---|---|
| 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) (handle) |
| IN | dest | rank of destination (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Issend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

Start a synchronous mode, nonblocking send.

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

| | | |
|---|---|---|
| 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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

Start a ready mode nonblocking send.

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

| | | |
|---|---|---|
| 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 or MPI_ANY_SOURCE (non-negative integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

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 modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

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

*Advice to users.* To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20. (*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 is also internally configured so that calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X return count = 0 and MPI_TEST_CANCELLED returns false. We set a status variable to empty when the value returned by it is not significant. Status is set in this way so as to prevent errors due to accesses of stale information.

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

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

MPI_WAIT(request, status)

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

**C binding**

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

**F08 binding**

```
MPI_Wait(request, status, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

A call to MPI_WAIT returns when the operation identified by request is complete. If the request is an active persistent request, it is marked inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. MPI_WAIT is a non-local 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 multithreaded 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 | communication request (handle) |
| OUT | flag | true if operation completed (logical) |
| OUT | status | status object (Status) |

**C binding**

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

**F08 binding**

```
MPI_Test(request, flag, status, ierror)
    TYPE(MPI_Request), INTENT(INOUT) ::  request
    LOGICAL, INTENT(OUT) ::  flag
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

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 request is an active persistent request, it is marked as inactive. Any other type of request is deallocated and the request handle is set to MPI_REQUEST_NULL. The call returns flag = false if the operation identified by request is not complete. 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.*)

**Example 3.11** 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 by using the following operation.

MPI_REQUEST_FREE(request)

| | | |
|---|---|---|
| INOUT | request | communication request (handle) |

**C binding**

```
int MPI_Request_free(MPI_Request *request)
```

**F08 binding**

```
MPI_Request_free(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

MPI_REQUEST_FREE is a local operation that marks the request object for deallocation and sets request to MPI_REQUEST_NULL. Ongoing communication, if any, that is associated with the request will be allowed to complete. The request will be deallocated only after its completion. Classes of operations described later in the standard, such as nonblocking collective and persistent collective (see Chapters 5 and 7), also use request objects. In the case of nonblocking collective operations and persistent collective operations, it is erroneous to call MPI_REQUEST_FREE unless the request is inactive.

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

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

**Example 3.12** An example using MPI_REQUEST_FREE.

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

### 3.7.4 Semantics of Nonblocking Communications

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

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

**Example 3.13** Message ordering for nonblocking operations.

```
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 3.14** An illustration of progress semantics.

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

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

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

### 3.7.5 Multiple Completions

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

MPI_WAITANY(count, array_of_requests, index, status)

| | | |
|---|---|---|
| IN | count | list length (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) |

**C binding**

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

**F08 binding**

```
MPI_Waitany(count, array_of_requests, index, status, ierror)
```

```
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: index
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

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 operation. (The array is indexed from zero in C, and from one in Fortran.) If the request is an active persistent request, it is marked inactive. Any other type of request 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 an 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)

| | | |
|---|---|---|
| IN | count | list length) (integer) |
| INOUT | array_of_requests | array of requests (array of handles) |
| OUT | index | index of operation that completed or MPI_UNDEFINED if none completed (integer) |
| OUT | flag | true if one of the operations is complete (logical) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_Testany(int count, MPI_Request array_of_requests[], int *index,
             int *flag, MPI_Status *status)
```

**F08 binding**

```
MPI_Testany(count, array_of_requests, index, flag, status, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: index
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

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

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

If the array of requests contains active handles then the execution of MPI_TESTANY(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.

MPI_WAITALL(count, array_of_requests, array_of_statuses)

| | | |
|---|---|---|
| IN | count | lists length (integer) |
| INOUT | array_of_requests | array of requests (array of handles) |
| OUT | array_of_statuses | array of status objects (array of Status) |

**C binding**

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],
             MPI_Status array_of_statuses[])
```

**F08 binding**

```
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Request), INTENT(INOUT) ::  array_of_requests(count)
    TYPE(MPI_Status) ::  array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

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. Active persistent requests are marked inactive. Requests of any other type are deallocated and the corresponding handles in the array are set to MPI_REQUEST_NULL. 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 (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) |

**C binding**

```
int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,
             MPI_Status array_of_statuses[])
```

**F08 binding**

```
MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
    LOGICAL FLAG
```

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

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

Errors that occurred during the execution of MPI_TESTALL are handled in the same manner as errors in MPI_WAITALL.

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

| | | |
|---|---|---|
| IN | incount | length of arry_of_requests (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) |

**C binding**

```
int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
             int *outcount, int array_of_indices[],
             MPI_Status array_of_statuses[])
```

**F08 binding**

```
MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
             array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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
```

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. Completed active persistent requests are marked as inactive. Any other type or request that completed 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 (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) |

**C binding**

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

**F08 binding**

```
MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
             array_of_statuses, ierror)
    INTEGER, INTENT(IN) ::  incount
    TYPE(MPI_Request), INTENT(INOUT) ::  array_of_requests(count)
    INTEGER, INTENT(OUT) ::  outcount, array_of_indices(*)
    TYPE(MPI_Status) ::  array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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
```

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 communications as possible. (*End of advice to implementors.*)

**Example 3.15** Client-server code (starvation can occur).

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

**Example 3.16** Same code, using MPI_WAITSOME.

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

MPI_REQUEST_GET_STATUS(request, flag, status)

| | | |
|---|---|---|
| IN | request | request (handle) |
| OUT | flag | boolean flag, same as from MPI_TEST (logical) |
| OUT | status | status object if flag is true (Status) |

**C binding**

```
int MPI_Request_get_status(MPI_Request request, int *flag,
             MPI_Status *status)
```

**F08 binding**

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

**F binding**

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

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

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

## 3.8 Probe and Cancel

The MPI_PROBE, MPI_IPROBE, MPI_MPROBE, and MPI_IMPROBE 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 MPI_CANCEL operation allows pending communications to be cancelled. 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.

Cancelling a send request by calling MPI_CANCEL is deprecated.

### 3.8.1 Probe

MPI_IPROBE(source, tag, comm, flag, status)

| | | |
|---|---|---|
| IN | source | rank of source or MPI_ANY_SOURCE (non-negative integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | flag | (logical) |
| OUT | status | status object (Status) |

**C binding**

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

**F08 binding**

```
MPI_Iprobe(source, tag, comm, flag, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

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 multithreaded, 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.

A probe with MPI_PROC_NULL as source returns flag = true, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 3.11.

MPI_PROBE(source, tag, comm, status)

| | | |
|---|---|---|
| IN | source | rank of source or MPI_ANY_SOURCE (non-negative integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

**F08 binding**

```
MPI_Probe(source, tag, comm, status, ierror)
    INTEGER, INTENT(IN) ::  source, tag
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

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

MPI_PROBE behaves like MPI_IPROBE except that it is a 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 or matched by a concurrent matched probe.

**Example 3.17**

Use blocking probe to wait for an incoming message.

```
       CALL MPI_COMM_RANK(comm, rank, ierr)
       IF (rank.EQ.0) THEN
           CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
       ELSE IF (rank.EQ.1) THEN
           CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
       ELSE IF (rank.EQ.2) THEN
           DO i=1, 2
              CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                             comm, status, ierr)
              IF (status(MPI_SOURCE) .EQ. 0) THEN
100               CALL MPI_RECV(i, 1, MPI_INTEGER, 0, 0, comm, status, ierr)
              ELSE
200               CALL MPI_RECV(x, 1, MPI_REAL, 1, 0, comm, status, ierr)
              END IF
           END DO
       END IF
```

Each message is received with the right type.

**Example 3.18** A similar program to the previous example, but now it has a problem.

```
       CALL MPI_COMM_RANK(comm, rank, ierr)
       IF (rank.EQ.0) THEN
            CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
       ELSE IF (rank.EQ.1) THEN
            CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
       ELSE IF (rank.EQ.2) THEN
           DO i=1, 2
              CALL MPI_PROBE(MPI_ANY_SOURCE, 0,
                              comm, status, ierr)
              IF (status(MPI_SOURCE) .EQ. 0) THEN
100                CALL MPI_RECV(i, 1, MPI_INTEGER, MPI_ANY_SOURCE,
                                 0, comm, status, ierr)
```

```
            ELSE
200             CALL MPI_RECV(x, 1, MPI_REAL, MPI_ANY_SOURCE,
                              0, comm, status, ierr)
            END IF
         END DO
      END IF
```

In Example 3.18, the two receive calls in statements labeled 100 and 200 in Example 3.17 slightly modified, using MPI_ANY_SOURCE as the source argument. The program is now incorrect: the receive operation may receive a message that is distinct from the message probed by the preceding call to MPI_PROBE.

*Advice to users.* In a multithreaded MPI program, MPI_PROBE and MPI_IPROBE might need special care. If a thread probes for a message and then immediately posts a matching receive, the receive may match a message other than that found by the probe since another thread could concurrently receive that original message [29]. MPI_MPROBE and MPI_IMPROBE solve this problem by matching the incoming message so that it may only be received with MPI_MRECV or MPI_IMRECV on the corresponding message handle. (*End of advice to users.*)

*Advice to implementors.* A call to MPI_PROBE(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.*)

### 3.8.2 Matching Probe

The function MPI_PROBE checks for incoming messages without receiving them. Since the list of incoming messages is global among the threads of each MPI process, it can be hard to use this functionality in threaded environments [29, 26].

Like MPI_PROBE and MPI_IPROBE, the MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message that was matched regardless of other intervening probe or receive operations. This gives the application an opportunity to decide how to receive the message, based on the information returned by the probe. In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.

MPI_IMPROBE(source, tag, comm, flag, message, status)

| | | |
|---|---|---|
| IN | source | rank of source or MPI_ANY_SOURCE (non-negative integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | flag | (logical) |
| OUT | message | returned message (handle) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
             MPI_Message *message, MPI_Status *status)
```

**F08 binding**

```
MPI_Improbe(source, tag, comm, flag, message, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Message), INTENT(OUT) :: message
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

MPI_IMPROBE(source, tag, comm, flag, message, 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. In addition, it returns in message a handle to the matched message. Otherwise, the call returns flag = false, and leaves status and message undefined.

A matched receive (MPI_MRECV or MPI_IMRECV) executed with the message handle will receive the message that was matched by the probe. Unlike MPI_IPROBE, no other probe or receive operation may match the message returned by MPI_IMPROBE. Each message returned by MPI_IMPROBE must be received with either MPI_MRECV or MPI_IMRECV.

The source argument of MPI_IMPROBE 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.

A synchronous send operation that is matched with MPI_IMPROBE or MPI_MPROBE will complete successfully only if both a matching receive is posted with MPI_MRECV or MPI_IMRECV, and the receive operation has started to receive the message sent by the synchronous send.

There is a special predefined message: MPI_MESSAGE_NO_PROC, which is a message which has MPI_PROC_NULL as its source process. The predefined constant MPI_MESSAGE_NULL is the value used for invalid message handles.

A matching probe with MPI_PROC_NULL as source returns flag = true, message = MPI_MESSAGE_NO_PROC, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0; see Section 3.11. It is not necessary to call MPI_MRECV or MPI_IMRECV with MPI_MESSAGE_NO_PROC, but it is not erroneous to do so.

*Rationale.* MPI_MESSAGE_NO_PROC was chosen instead of MPI_MESSAGE_PROC_NULL to avoid possible confusion as another null handle constant. (*End of rationale.*)

MPI_MPROBE(source, tag, comm, message, status)

| | | |
|---|---|---|
| IN | source | rank of source or MPI_ANY_SOURCE (non-negative integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | message | returned message (handle) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,
             MPI_Status *status)
```

**F08 binding**

```
MPI_Mprobe(source, tag, comm, message, status, ierror)
    INTEGER, INTENT(IN) ::  source, tag
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Message), INTENT(OUT) ::  message
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_MPROBE(SOURCE, TAG, COMM, MESSAGE, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
```

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

The implementation of MPI_MPROBE and MPI_IMPROBE needs to guarantee progress in the same way as in the case of MPI_PROBE and MPI_IPROBE.

### 3.8.3 Matched Receives

The functions MPI_MRECV and MPI_IMRECV receive messages that have been previously matched by a matching probe (Section 3.8.2).

MPI_MRECV(buf, count, datatype, message, 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) |
| INOUT | message | message (handle) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype,
              MPI_Message *message, MPI_Status *status)
```

**F08 binding**

```
MPI_Mrecv(buf, count, datatype, message, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Message), INTENT(INOUT) :: message
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
```

This call receives a message matched by a matching probe operation (Section 3.8.2).

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

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

If MPI_MRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the call returns immediately with the status object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0, as if a receive from MPI_PROC_NULL was issued (see Section 3.11). A call to MPI_MRECV with MPI_MESSAGE_NULL is erroneous.

MPI_IMRECV(buf, count, datatype, message, request)

| | | |
|---|---|---|
| OUT | buf | initial address of receive buffer (choice) |
| IN | count | number of elements in receive buffer (non-negative integer) |
| IN | datatype | datatype of each receive buffer element (handle) |
| INOUT | message | message (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,
             MPI_Message *message, MPI_Request *request)
```

**F08 binding**

```
MPI_Imrecv(buf, count, datatype, message, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Message), INTENT(INOUT) :: message
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR
```

MPI_IMRECV is the nonblocking variant of MPI_MRECV and starts a nonblocking receive of a matched message. Completion semantics are similar to MPI_IRECV as described in Section 3.7.2. On return from this function, the message handle is set to MPI_MESSAGE_NULL.

If MPI_IMRECV is called with MPI_MESSAGE_NO_PROC as the message argument, the call returns immediately with a request object which, when completed, will yield a status object set to source = MPI_PROC_NULL, tag = MPI_ANY_TAG, and count = 0, as if a receive from MPI_PROC_NULL was issued (see Section 3.11). A call to MPI_IMRECV with MPI_MESSAGE_NULL is erroneous.

> *Advice to implementors.* If reception of a matched message is started with MPI_IMRECV, then it is possible to cancel the returned request with MPI_CANCEL. If MPI_CANCEL succeeds, the matched message must be found by a subsequent message probe (MPI_PROBE, MPI_IPROBE, MPI_MPROBE, or MPI_IMPROBE), received by a subsequent receive operation or cancelled by the sender. See Section 3.8.4 for details about MPI_CANCEL. The cancellation of operations initiated with MPI_IMRECV may fail. (*End of advice to implementors.*)

### 3.8.4 Cancel

MPI_CANCEL(request)

| | | |
|---|---|---|
| IN | request | communication request (handle) |

**C binding**

```
int MPI_Cancel(MPI_Request *request)
```

**F08 binding**

```
MPI_Cancel(request, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

A call to MPI_CANCEL marks for cancellation a pending, nonblocking communication operation (send or receive). Cancelling a send request by calling MPI_CANCEL is deprecated. The cancel call is local. It returns immediately, possibly before the communication is actually cancelled. It is still necessary to call MPI_REQUEST_FREE, MPI_WAIT or MPI_TEST (or any of the derived operations) with the cancelled request as argument after the call to MPI_CANCEL. 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 cancelled communication, then MPI_TEST will eventually be successful.

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. Cancelling a persistent send request by calling MPI_CANCEL is deprecated. A successful cancellation cancels the active communication, but not the request itself. After the call to MPI_CANCEL and the subsequent call to MPI_WAIT or MPI_TEST, the request becomes inactive and can be activated for a new communication.

The successful cancellation of a buffered send frees the buffer space occupied by the pending message. Cancelling a buffered send request by calling MPI_CANCEL is deprecated.

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

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

*Rationale.* Although the IN request handle parameter should not need to be passed by reference, the C binding has listed the argument type as MPI_Request* since MPI-

1.0. This function signature therefore cannot be changed without breaking existing MPI applications. (*End of rationale.*)

MPI_TEST_CANCELLED(status, flag)

| | | |
|---|---|---|
| IN | status | status object (Status) |
| OUT | flag | (logical) |

**C binding**

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

**F08 binding**

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

**F binding**

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

Returns flag = true if the communication associated with the status object was cancelled 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 cancelled then one should call MPI_TEST_CANCELLED first, to check whether the operation was cancelled, before checking on the other fields of the return status.

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

*Advice to implementors.* If a send operation uses an "eager" protocol (data is transferred to the receiver before a matching receive is posted), then the cancellation of this send may require communication with the intended receiver in order to free allocated buffers. On some systems this may require an interrupt to the intended receiver. Note that, while communication may be needed to implement MPI_CANCEL, this is still a local 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 (with the exception of the buffer contents) 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 operations. In the case of point-to-point communication, 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 point-to-point request be received by a receive operation using a persistent point-to-point request, or vice versa.

There are also collective communication persistent operations defined in Section 5.13 and Section 7.8. The remainder of this section covers the point-to-point persistent initialization operations and the start routines, which are used for both point-to-point and collective persistent communication.

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

MPI_SEND_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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

Creates a persistent communication request for a standard mode send operation, and binds to it all the arguments of a send operation.

MPI_BSEND_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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
```

```
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

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 (non-negative integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

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 (non-negative integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

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

**F08 binding**

```
MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

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) |

**C binding**

```
int MPI_Start(MPI_Request *request)
```

**F08 binding**

```
MPI_Start(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

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 modified 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 MPI_ISEND; a call to MPI_START with a request created by MPI_BSEND_INIT starts a communication in the same manner as a call to MPI_IBSEND; and so on.

MPI_STARTALL(count, array_of_requests)

| | | |
|---|---|---|
| IN | count | list length (integer) |
| INOUT | array_of_requests | array of requests (array of handles) |

**C binding**

```
int MPI_Startall(int count, MPI_Request array_of_requests[])
```

**F08 binding**

```
MPI_Startall(count, array_of_requests, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR
```

Start all communications associated with requests in array_of_requests. A call to MPI_STARTALL(count, array_of_requests) has the same effect as calls to MPI_START (&array_of_requests[i]), executed for i=0 ,..., count-1, in some arbitrary order.

A communication started with a call to MPI_START or MPI_STARTALL is completed by a call to MPI_WAIT, MPI_TEST, or one of the derived functions described in Section 3.7.5. The request becomes inactive after successful completion of such call. The request is not deallocated and it can be activated anew by an MPI_START or MPI_STARTALL call.

A persistent request is deallocated by a call to MPI_REQUEST_FREE (Section 3.7.3).

The call to MPI_REQUEST_FREE can occur at any point in the program after the persistent request was created. However, the request will be deallocated only after it becomes inactive. Active receive requests should not be freed. Otherwise, it will not be possible to check that the receive has completed. Collective operation requests (defined in Section 5.12 and Section 7.7 for nonblocking collective operations, and Section 5.13 and Section 7.8 for persistent collective operations) must not be freed while active. It is preferable, in general, to free requests when they are inactive. If this rule is followed, then the functions described in this section will be invoked in a sequence of the form,

**Create (Start Complete)$^*$ Free**

where $*$ indicates zero or more repetitions. If the same 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 MPI_START can be matched with any receive operation and, likewise, a receive operation initiated with MPI_START can receive messages generated by any send operation.

*Advice to users.* To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20. (*End of advice to users.*)

## 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(sendbuf, 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 (non-negative 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 receive buffer element (handle) |
| IN | source | rank of source or MPI_ANY_SOURCE (non-negative integer) |
| IN | recvtag | receive tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_Sendrecv(const 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)
```

**F08 binding**

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcount, dest, sendtag, recvcount, source,
    recvtag
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

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 (non-negative integer) |
| IN | sendtag | send message tag (integer) |
| IN | source | rank of source or MPI_ANY_SOURCE (non-negative integer) |
| IN | recvtag | receive message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | status | status object (Status) |

**C binding**

```
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)
```

**F08 binding**

```
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
             comm, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, dest, sendtag, source, recvtag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR
```

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. A probe or matching probe with source = MPI_PROC_NULL succeeds and returns as soon as possible, and the status object returns source = MPI_PROC_NULL, tag = MPI_ANY_TAG and count = 0. A matching probe (cf. Section 3.8.2) with MPI_PROC_NULL as source returns flag = true, message = MPI_MESSAGE_NO_PROC, and 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 and in Section 3.3. In this chapter, this model is extended to describe any data layout. We consider general datatypes that allow one to transfer efficiently heterogeneous and noncontiguous data. We conclude with the description of calls for explicit packing and unpacking of messages.

## 4.1 Derived Datatypes

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

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

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

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

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

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

Let

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

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

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

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 $\mathsf{buf} + disp_i$ and has type $type_i$. A message assembled from such a communication buffer will consist of $n$ values, of the types defined by $Typesig$.

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

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 MPI_SEND(buf, 1, datatype,...) will use the send buffer defined by the base address buf and the general datatype associated with datatype; it will generate a message with the type signature determined by the datatype argument. MPI_RECV(buf, 1, datatype,...) will use the receive buffer defined by the base address buf and the general datatype associated with 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, MPI_INT is a predefined handle to a datatype with type map $\{(\texttt{int}, 0)\}$, with one entry of type `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

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

then

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + \mathsf{sizeof}(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{4.1}$$

If $type_j$ requires alignment to a byte address that is a multiple of $k_j$, then $\epsilon$ is the least non-negative increment needed to round $extent(Typemap)$ to the next multiple of $\max_j k_j$. In Fortran, it is implementation dependent whether the MPI implementation computes the alignments $k_j$ according to the alignments used by the compiler in common blocks, `SEQUENCE` derived types, `BIND(C)` derived types, or derived types that are neither `SEQUENCE` nor `BIND(C)`. The complete definition of **extent** is given by Equation 4.1 Section 4.1.

**Example 4.1** Assume that $Type = \{(\texttt{double}, 0), (\texttt{char}, 8)\}$ (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. In Fortran, structures can be expressed with several language features, e.g., common blocks, `SEQUENCE` derived types, or `BIND(C)` derived types. The compiler may use different alignments, and therefore, it is recommended to use MPI_TYPE_CREATE_RESIZED for arrays of structures if an alignment may cause an alignment-gap at the end of a structure as described in Section 4.1.6 and in Section 17.1.15. (*End of rationale.*)

### 4.1.1 Type Constructors with Explicit Addresses

In Fortran, the functions MPI_TYPE_CREATE_HVECTOR, MPI_TYPE_CREATE_HINDEXED, MPI_TYPE_CREATE_HINDEXED_BLOCK, MPI_TYPE_CREATE_STRUCT, and MPI_GET_ADDRESS accept arguments of type `INTEGER(KIND=MPI_ADDRESS_KIND)`, wherever arguments of type MPI_Aint are used in C. 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 MPI_TYPE_CONTIGUOUS which allows replication of a datatype into contiguous locations.

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

| | | |
|---|---|---|
| IN | count | replication count (integer) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

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

**F08 binding**

```
MPI_Type_contiguous(count, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

newtype 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 $\{(\texttt{double}, 0), (\texttt{char}, 8)\}$, with extent 16, and let count $= 3$. The type map of the datatype returned by newtype is

$$\{(\texttt{double}, 0), (\texttt{char}, 8), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{double}, 32), (\texttt{char}, 40)\};$$

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 $\mathsf{count} \cdot \mathsf{n}$ entries defined by:

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

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

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.

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

| | | |
|---|---|---|
| IN | count | number of blocks (integer) |
| IN | blocklength | number of elements in each block (non-negative integer) |
| IN | stride | number of elements between start of each block (integer) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_vector(int count, int blocklength, int stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count, blocklength, stride
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

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

$$\{(\texttt{double}, 0), (\texttt{char}, 8), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{double}, 32), (\texttt{char}, 40),$$

$$(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104)\}.$$

That is, two blocks with three copies each of the old type, with a stride of 4 elements ($4 \cdot 16$ bytes) between the the start of each block.

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

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

In general, assume that oldtype has type map,

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

with extent $ex$. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot \mathsf{n}$ entries:

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

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

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

$$(type_0, disp_0 + \mathsf{stride} \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot ex), \ldots,$$

$$(type_0, disp_0 + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + (\mathsf{stride} + \mathsf{bl} - 1) \cdot ex), \ldots,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), \ldots,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) \cdot ex), \ldots,$$

$$(type_0, disp_0 + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex), \ldots,$$

$$(type_{n-1}, disp_{n-1} + (\mathsf{stride} \cdot (\mathsf{count} - 1) + \mathsf{bl} - 1) \cdot ex)\}.$$

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)

| | | |
|---|---|---|
| IN | count | number of blocks (integer) |
| IN | blocklength | number of elements in each block (non-negative integer) |
| IN | stride | number of bytes between start of each block (integer) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride,
             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
             ierror)
    INTEGER, INTENT(IN) :: count, blocklength
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
             IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) STRIDE
```

Assume that oldtype has type map,

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

with extent $ex$. Let bl be the blocklength. The newly created datatype has a type map with $\mathsf{count} \cdot \mathsf{bl} \cdot n$ entries:

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

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

$$(type_0, disp_0 + (\mathsf{bl} - 1) \cdot ex), \ldots, (type_{n-1}, disp_{n-1} + (\mathsf{bl} - 1) \cdot ex),$$

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

$$(type_0, disp_0 + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), \ldots,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} + (\mathsf{bl} - 1) \cdot ex), \ldots,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1)), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1)), \ldots,$$

$$(type_0, disp_0 + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex), \ldots,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{stride} \cdot (\mathsf{count} - 1) + (\mathsf{bl} - 1) \cdot ex)\}.$$

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.

MPI_TYPE_INDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

| | | |
|---|---|---|
| IN | count | number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (integer) |
| IN | array_of_blocklengths | number of elements per block (array of non-negative integers) |
| IN | array_of_displacements | displacement for each block, in multiples of oldtype (array of integers) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_indexed(int count, const int array_of_blocklengths[],
             const int array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(count),
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

**Example 4.5**

Let oldtype have type map $\{(\texttt{double}, 0), (\texttt{char}, 8)\}$, with extent 16. Let B = (3, 1) and let D = (4, 0). A call to MPI_TYPE_INDEXED(2, B, D, oldtype, newtype) returns a datatype with type map,

$$\{(\texttt{double}, 64), (\texttt{char}, 72), (\texttt{double}, 80), (\texttt{char}, 88), (\texttt{double}, 96), (\texttt{char}, 104),$$
$$(\texttt{double}, 0), (\texttt{char}, 8)\}.$$

That is, three copies of the old type starting at displacement 64, and one copy starting at displacement 0.

In general, assume that oldtype has type map,

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

with extent $ex$. Let B be the array_of_blocklengths argument and D be the array_of_displacements argument. The newly created datatype has $n \cdot \sum_{i=0}^{\mathsf{count}-1} \mathsf{B[i]}$ entries:

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

A call to MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype) is equivalent to a call to MPI_TYPE_INDEXED(count, B, D, oldtype, newtype) where

$$\mathsf{D[j]} = j \cdot \mathsf{stride},\ j = 0, \ldots, \mathsf{count} - 1,$$

and

$$\mathsf{B[j]} = \mathsf{blocklength},\ j = 0, \ldots, \mathsf{count} - 1.$$

**Hindexed** The function MPI_TYPE_CREATE_HINDEXED is identical to MPI_TYPE_INDEXED, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.

MPI_TYPE_CREATE_HINDEXED(count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

| | | |
|---|---|---|
| IN | count | number of blocks – also number of entries in array_of_displacements and array_of_blocklengths (integer) |
| IN | array_of_blocklengths | number of elements in each block (array of non-negative integers) |
| IN | array_of_displacements | byte displacement of each block (array of integers) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

### C binding

```
int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],
             const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_hindexed(count, array_of_blocklengths,
             array_of_displacements, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(count)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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(*)
```

Assume that oldtype has type map,

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

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

$$\{(type_0, disp_0 + \mathsf{D[0]}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{D[0]}), \ldots,$$

$$(type_0, disp_0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), \ldots,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex), \ldots,$$

$$(type_0, disp_0 + \mathsf{D[count\text{-}1]}), \ldots, (type_{n-1}, disp_{n-1} + \mathsf{D[count\text{-}1]}), \ldots,$$

$$(type_0, disp_0 + \mathsf{D[count\text{-}1]} + (\mathsf{B[count\text{-}1]} - 1) \cdot ex), \ldots,$$

$$(type_{n-1}, disp_{n-1} + \mathsf{D[count\text{-}1]} + (\mathsf{B[count\text{-}1]} - 1) \cdot ex)\}.$$

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

MPI_TYPE_CREATE_INDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype, newtype)

| | | |
|---|---|---|
| IN | count | length of array of displacements (integer) |
| IN | blocklength | size of block (non-negative integer) |
| IN | array_of_displacements | array of displacements (array of integers) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_create_indexed_block(int count, int blocklength,
             const int array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, blocklength,
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_CREATE_INDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,
    NEWTYPE, IERROR
```

Hindexed_block The function MPI_TYPE_CREATE_HINDEXED_BLOCK is identical to MPI_TYPE_CREATE_INDEXED_BLOCK, except that block displacements in array_of_displacements are specified in bytes, rather than in multiples of the oldtype extent.

MPI_TYPE_CREATE_HINDEXED_BLOCK(count, blocklength, array_of_displacements, oldtype, newtype)

| | | |
|---|---|---|
| IN | count | length of array of displacements (integer) |
| IN | blocklength | size of block (non-negative integer) |
| IN | array_of_displacements | byte displacement of each block (array of integers) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_create_hindexed_block(int count, int blocklength,
             const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, blocklength
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_CREATE_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

Struct MPI_TYPE_CREATE_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)

| | | |
|---|---|---|
| IN | count | number of blocks – also number of entries in arrays array_of_types, array_of_displacements, and array_of_blocklengths (integer) |
| IN | array_of_blocklengths | number of elements in each block (array of non-negative integers) |
| IN | array_of_displacements | byte displacement of each block (array of integers) |
| IN | array_of_types | types of elements in each block (array of handles) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
             const MPI_Aint array_of_displacements[],
             const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_struct(count, array_of_blocklengths,
             array_of_displacements, array_of_types, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(count)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) ::  array_of_types(count)
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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,
    IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)
```

**Example 4.6** Let type1 have type map,

$$\{(\texttt{double}, 0), (\texttt{char}, 8)\},$$

with extent 16. Let B = (2, 1, 3), D = (0, 16, 26), and T = (MPI_FLOAT, type1, MPI_CHAR). Then a call to MPI_TYPE_CREATE_STRUCT(3, B, D, T, newtype) returns a datatype with type map,

$$\{(\texttt{float}, 0), (\texttt{float}, 4), (\texttt{double}, 16), (\texttt{char}, 24), (\texttt{char}, 26), (\texttt{char}, 27), (\texttt{char}, 28)\}.$$

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.)

In general, let T be the array_of_types argument, where T[i] is a handle to,

$$typemap_i = \{(type_0^i, disp_0^i), \ldots, (type_{n_i-1}^i, disp_{n_i-1}^i)\},$$

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 $\sum_{i=0}^{\mathsf{c}-1} \mathsf{B[i]} \cdot n_i$ entries:

$$\{(type_0^0, disp_0^0 + \mathsf{D[0]}), \ldots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D[0]}), \ldots,$$

$$(type_0^0, disp_0^0 + \mathsf{D[0]} + (\mathsf{B[0]} - 1) \cdot ex_0), \ldots, (type_{n_0}^0, disp_{n_0}^0 + \mathsf{D[0]} + (\mathsf{B[0]\text{-}1}) \cdot ex_0), \ldots,$$

$$(type_0^{\mathsf{c}-1}, disp_0^{\mathsf{c}-1} + \mathsf{D[c\text{-}1]}), \ldots, (type_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1}, disp_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1} + \mathsf{D[c\text{-}1]}), \ldots,$$

$$(type_0^{\mathsf{c}-1}, disp_0^{\mathsf{c}-1} + \mathsf{D[c\text{-}1]} + (\mathsf{B[c\text{-}1]} - 1) \cdot ex_{\mathsf{c}-1}), \ldots,$$

$$(type_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1}, disp_{n_{\mathsf{c}-1}-1}^{\mathsf{c}-1} + \mathsf{D[c\text{-}1]} + (\mathsf{B[c\text{-}1]\text{-}1}) \cdot ex_{\mathsf{c}-1})\}.$$

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 (integer) |
| IN | array_of_sizes | number of elements of type oldtype in each dimension of the full array (array of non-negative integers) |
| IN | array_of_subsizes | number of elements of type oldtype in each dimension of the subarray (array of non-negative integers) |
| IN | array_of_starts | starting coordinates of the subarray in each dimension (array of non-negative integers) |
| IN | order | array storage order flag (state) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
             const int array_of_subsizes[], const int array_of_starts[],
             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
             array_of_starts, order, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
    array_of_subsizes(ndims), array_of_starts(ndims), order
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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
```

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.

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 ndims parameter specifies the number of dimensions in the full data array and gives the number of elements in array_of_sizes, array_of_subsizes, and array_of_starts.

The number of elements of type oldtype in each dimension of the $n$-dimensional array and the requested subarray are specified by array_of_sizes and array_of_subsizes, respectively. For any dimension i, it is erroneous to specify array_of_subsizes[i] $< 1$ or array_of_subsizes[i] $>$ array_of_sizes[i].

The 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 array_of_starts[i] $< 0$ or array_of_starts[i] $>$ (array_of_sizes[i] $-$ 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 array_of_starts for that dimension is n-1. (*End of advice to users.*)

The order argument specifies the storage order for the subarray as well as the full array. It must be set to one of the following:

**MPI_ORDER_C** The ordering used by C arrays, (i.e., row-major order)

**MPI_ORDER_FORTRAN** The ordering used by Fortran arrays, (i.e., column-major order)

A ndims-dimensional subarray (newtype) with no extra padding can be defined by the function Subarray() as follows:

$$\begin{aligned} \mathsf{newtype} \;=\;& \text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\ & \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\ & \{start_0, start_1, \ldots, start_{ndims-1}\}, \mathsf{oldtype}) \end{aligned}$$

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. Then we define the Subarray() function recursively using the following three equations. Equation 4.2 defines the base step. Equation 4.3 defines the recursion step when order = MPI_ORDER_FORTRAN, and Equation 4.4 defines the recursion step when order = MPI_ORDER_C. These equations use the conceptual datatypes lb_marker and ub_marker, see Section 4.1.6 for details.

$$\begin{aligned} & \text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \\ & \qquad \{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}) \\ & = \{(\mathsf{lb\_marker}, 0), \\ & \quad (type_0, disp_0 + start_0 \times ex), \ldots, (type_{n-1}, disp_{n-1} + start_0 \times ex), \\ & \quad (type_0, disp_0 + (start_0 + 1) \times ex), \ldots, (type_{n-1}, \\ & \qquad disp_{n-1} + (start_0 + 1) \times ex), \ldots \\ & \quad (type_0, disp_0 + (start_0 + subsize_0 - 1) \times ex), \ldots, \\ & \qquad (type_{n-1}, disp_{n-1} + (start_0 + subsize_0 - 1) \times ex), \\ & \quad (\mathsf{ub\_marker}, size_0 \times ex)\} \end{aligned} \tag{4.2}$$

$$\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \tag{4.3}$$

$$\{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\},$$
$$\{start_0, start_1, \ldots, start_{ndims-1}\}, \textsf{oldtype})$$
$$= \text{Subarray}(ndims-1, \{size_1, size_2, \ldots, size_{ndims-1}\},$$
$$\{subsize_1, subsize_2, \ldots, subsize_{ndims-1}\},$$
$$\{start_1, start_2, \ldots, start_{ndims-1}\},$$
$$\text{Subarray}(1, \{size_0\}, \{subsize_0\}, \{start_0\}, \textsf{oldtype}))$$

$$\begin{aligned} &\text{Subarray}(ndims, \{size_0, size_1, \ldots, size_{ndims-1}\}, \\ &\quad \{subsize_0, subsize_1, \ldots, subsize_{ndims-1}\}, \\ &\quad \{start_0, start_1, \ldots, start_{ndims-1}\}, \textsf{oldtype}) \\ &= \text{Subarray}(ndims-1, \{size_0, size_1, \ldots, size_{ndims-2}\}, \\ &\quad \{subsize_0, subsize_1, \ldots, subsize_{ndims-2}\}, \\ &\quad \{start_0, start_1, \ldots, start_{ndims-2}\}, \\ &\qquad \text{Subarray}(1, \{size_{ndims-1}\}, \{subsize_{ndims-1}\}, \{start_{ndims-1}\}, \textsf{oldtype})) \end{aligned} \tag{4.4}$$

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 [42] 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 and Section 13.3. 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 (non-negative integer) |
| IN | rank | rank in process group (non-negative integer) |
| IN | ndims | number of array dimenstions as well as process grid dimensions (integer) |
| IN | array_of_gsizes | number of elements of type oldtype in each dimension of global array (array of non-negative integers) |
| IN | array_of_distribs | distribution of array in each dimension (array of non-negative integers) |
| IN | array_of_dargs | distribution argument in each dimension (array of non-negative integers) |
| IN | array_of_psizes | size of process grid in each dimension (array of non-negative integers) |
| IN | order | array storage order flag (state) |
| IN | oldtype | old datatype (handle) |
| OUT | newtype | new datatype (handle) |

**C binding**

```
int MPI_Type_create_darray(int size, int rank, int ndims,
             const int array_of_gsizes[], const int array_of_distribs[],
             const int array_of_dargs[], const int array_of_psizes[],
             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
             array_of_distribs, array_of_dargs, array_of_psizes, order,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  size, rank, ndims, array_of_gsizes(ndims),
    array_of_distribs(ndims), array_of_dargs(ndims),
    array_of_psizes(ndims), order
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
             ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES, ORDER,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
    ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

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.) For a call to MPI_TYPE_CREATE_DARRAY to be correct, the equation $\prod_{i=0}^{ndims-1} array\_of\_psizes[i] = size$ must be satisfied. The ordering of processes in the process grid is assumed to be row-major, as in the case of virtual Cartesian process topologies.

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

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 array_of_dargs[i] $*$ array_of_psizes[i] $<$ array_of_gsizes[i].

For example, the HPF layout `ARRAY(CYCLIC(15))` corresponds to MPI_DISTRIBUTE_CYCLIC with a distribution argument of 15, and the HPF layout 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.

MPI_DISTRIBUTE_BLOCK with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to

$$(\text{array\_of\_gsizes[i]} + \text{array\_of\_psizes[i]} - 1)/\text{array\_of\_psizes[i]}.$$

If array_of_dargs[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 array_of_dargs[i] set to array_of_gsizes[i].

Finally, MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] equal to MPI_DISTRIBUTE_DFLT_DARG is equivalent to MPI_DISTRIBUTE_CYCLIC with array_of_dargs[i] set to 1.

For MPI_ORDER_FORTRAN, an ndims-dimensional distributed array (newtype) is defined by the following code fragment:

```
oldtypes[0] = oldtype;
for (i = 0; i < ndims; i++) {
    oldtypes[i+1] = cyclic(array_of_dargs[i],
                           array_of_gsizes[i],
                           r[i],
                           array_of_psizes[i],
                           oldtypes[i]);
}
newtype = oldtypes[ndims];
```

For MPI_ORDER_C, the code is:

```
oldtypes[0] = oldtype;
for (i = 0; i < ndims; i++) {
    oldtypes[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],
                             oldtypes[i]);
}
newtype = oldtypes[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:

```
t_rank = rank;
t_size = 1;
for (i = 0; i < ndims; i++)
    t_size *= array_of_psizes[i];
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. The following function uses the conceptual datatypes lb_marker and ub_marker, see Section 4.1.6 for details.

Given the above, the function cyclic() is defined as follows:

$$\begin{aligned} &\text{cyclic}(darg, gsize, r, psize, \mathsf{oldtype}) \\ &\quad = \{(\mathsf{lb\_marker}, 0), \\ &\qquad (type_0, disp_0 + r \times darg \times ex), \ldots, \\ &\qquad\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex), \end{aligned}$$

$$
\begin{aligned}
&(type_0, disp_0 + (r \times darg + 1) \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex), \\
&\ldots \\
&(type_0, disp_0 + ((r + 1) \times darg - 1) \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + ((r + 1) \times darg - 1) \times ex), \\
\\
&(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex), \\
&(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex + psize \times darg \times ex), \\
&\ldots \\
&(type_0, disp_0 + ((r + 1) \times darg - 1) \times ex + psize \times darg \times ex), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + ((r + 1) \times darg - 1) \times ex + psize \times darg \times ex), \\
&\qquad \vdots \\
&(type_0, disp_0 + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + r \times darg \times ex + psize \times darg \times ex \times (count - 1)), \\
&(type_0, disp_0 + (r \times darg + 1) \times ex + psize \times darg \times ex \times (count - 1)), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + 1) \times ex \\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)), \\
&\ldots \\
&(type_0, disp_0 + (r \times darg + darg_{last} - 1) \times ex \\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)), \ldots, \\
&\qquad (type_{n-1}, disp_{n-1} + (r \times darg + darg_{last} - 1) \times ex \\
&\qquad\qquad + psize \times darg \times ex \times (count - 1)), \\
&(\mathsf{ub\_marker}, gsize * ex)\}
\end{aligned}
$$

where *count* is defined by this code fragment:

```
nblocks = (gsize + (darg - 1)) / darg;
count = nblocks / psize;
left_over = nblocks - count * psize;
if (r < left_over)
    count = count + 1;
```

Here, *nblocks* is the number of blocks that must be distributed among the processors. Finally, $darg_{last}$ is defined by this code fragment:

```
if ((num_in_last_cyclic = gsize % (psize * darg)) == 0)
    darg_last = darg;
else {
    darg_last = num_in_last_cyclic - darg * r;
    if (darg_last > darg)
        darg_last = darg;
    if (darg_last <= 0)
```

```
            darg_last = darg;
        }
```

**Example 4.7** Consider generating the filetypes corresponding to the HPF distribution:

```
      <oldtype> FILEARRAY(100, 200, 300)
!HPF$ PROCESSORS PROCESSES(2, 3)
!HPF$ DISTRIBUTE FILEARRAY(CYCLIC(10), *, BLOCK) ONTO PROCESSES
```

This can be achieved by the following Fortran code, assuming there will be six processes attached to the run:

```
    ndims = 3
    array_of_gsizes(1) = 100
    array_of_distribs(1) = MPI_DISTRIBUTE_CYCLIC
    array_of_dargs(1) = 10
    array_of_gsizes(2) = 200
    array_of_distribs(2) = MPI_DISTRIBUTE_NONE
    array_of_dargs(2) = 0
    array_of_gsizes(3) = 300
    array_of_distribs(3) = MPI_DISTRIBUTE_BLOCK
    array_of_dargs(3) = MPI_DISTRIBUTE_DFLT_DARG
    array_of_psizes(1) = 2
    array_of_psizes(2) = 1
    array_of_psizes(3) = 3
    call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, array_of_gsizes, &
         array_of_distribs, array_of_dargs, array_of_psizes,        &
         MPI_ORDER_FORTRAN, oldtype, newtype, ierr)
```

### 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. Note that in Fortran MPI_BOTTOM is not usable for initialization or assignment, see Section 2.5.4.

The address of a location in memory can be found by invoking the function MPI_GET_ADDRESS. The **relative displacement** between two absolute addresses can be calculated with the function MPI_AINT_DIFF. A new absolute address as sum of an absolute base address and a relative displacement can be calculated with the function MPI_AINT_ADD. To ensure portability, arithmetic on absolute addresses should not be performed with the intrinsic operators “-” and “+”. See also Sections 2.5.6 and 4.1.12 on pages 16 and 121.

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

MPI_GET_ADDRESS(location, address)

| | | |
|---|---|---|
| IN | location | location in caller memory (choice) |
| OUT | address | address of location (integer) |

**C binding**

```
int MPI_Get_address(const void *location, MPI_Aint *address)
```

**F08 binding**

```
MPI_Get_address(location, address, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  location
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  address
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
    INTEGER IERROR
```

Returns the (byte) address of location.

*Rationale.* In the mpi_f08 module, the location argument is not defined with INTENT(IN) because existing applications may use MPI_GET_ADDRESS as a substitute for MPI_F_SYNC_REG that was not defined before MPI-3.0. (*End of rationale.*)

**Example 4.8** Using MPI_GET_ADDRESS for an array.

```
   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 = MPI_AINT_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 Sections 17.1.10–17.1.20. (*End of advice to users.*)

To ensure portability, arithmetic on MPI addresses must be performed using the MPI_AINT_ADD and MPI_AINT_DIFF functions.

MPI_AINT_ADD(base, disp)

| | | |
|---|---|---|
| IN | base | base address (integer) |
| IN | disp | displacement (integer) |

**C binding**

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
```

**F08 binding**

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  base, disp
```

**F binding**

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)
    INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP
```

MPI_AINT_ADD produces a new MPI_Aint value that is equivalent to the sum of the base and disp arguments, where base represents a base address returned by a call to MPI_GET_ADDRESS and disp represents a signed integer displacement. The resulting address is valid only at the process that generated base, and it must correspond to a location in the same object referenced by base, as described in Section 4.1.12. The addition is performed in a manner that results in the correct MPI_Aint representation of the output address, as if the process that originally produced base had called:

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

MPI_AINT_DIFF(addr1, addr2)

| | | |
|---|---|---|
| IN | addr1 | minuend address (integer) |
| IN | addr2 | subtrahend address (integer) |

**C binding**

```
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)
```

**F08 binding**

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  addr1, addr2
```

**F binding**

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

MPI_AINT_DIFF produces a new MPI_Aint value that is equivalent to the difference between addr1 and addr2 arguments, where addr1 and addr2 represent addresses returned by calls to MPI_GET_ADDRESS. The resulting address is valid only at the process that generated addr1 and addr2, and addr1 and addr2 must correspond to locations in the same object in the same process, as described in Section 4.1.12. The difference is calculated in a manner that results in the signed difference from addr1 to addr2, as if the process that originally produced the addresses had called (char *) addr1 - (char *) addr2 on the addresses initially passed to MPI_GET_ADDRESS.

The following auxiliary functions provide useful information on derived datatypes.

MPI_TYPE_SIZE(datatype, size)

| | | |
|---|---|---|
| IN | datatype | datatype (handle) |
| OUT | size | datatype size (non-negative integer) |

**C binding**

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

**F08 binding**

```
MPI_Type_size(datatype, size, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

MPI_TYPE_SIZE_X(datatype, size)

| | | |
|---|---|---|
| IN | datatype | datatype (handle) |
| OUT | size | datatype size (integer) |

**C binding**

```
int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)
```

**F08 binding**

```
MPI_Type_size_x(datatype, size, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_SIZE_X(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, IERROR
```

```
INTEGER(KIND=MPI_COUNT_KIND) SIZE
```

MPI_TYPE_SIZE and MPI_TYPE_SIZE_X set the value of size to 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. For both functions, if the OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

### 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 110. 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 overide the alignment rules that are used to compute upper bounds and extents. E.g., a C compiler may allow the user to overide default alignment rules for some of the structures within a program. The user has to specify explicitly the bounds of the datatypes that match these structures.

To achieve this, we add two additional conceptual datatypes, **lb_marker** and **ub_marker**, that represent the lower bound and upper bound of a datatype. These conceptual datatypes occupy no space ($extent(\text{lb\_marker}) = extent(\text{ub\_marker}) = 0$) . 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** A call to MPI_TYPE_CREATE_RESIZED(MPI_INT, -3, 9, type1) creates a new datatype that has an extent of 9 (from -3 to 5, 5 included), and contains an integer at displacement 0. This is the datatype defined by the typemap {(lb_marker, -3), (int, 0), (ub_marker, 6)}. If this type is replicated twice by a call to MPI_TYPE_CONTIGUOUS(2, type1, type2) then the newly created type can be described by the typemap {(lb_marker, -3), (int, 0), (int,9), (ub_marker, 15)}. (An entry of type ub_marker can be deleted if there is another entry of type ub_marker with a higher displacement; an entry of type lb_marker can be deleted if there is another entry of type lb_marker with a lower displacement.)

In general, if

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

then the **lower bound** of *Typemap* is defined to be

$$lb(Typemap) = \begin{cases} \min_j disp_j & \text{if no entry has type lb\_marker} \\ \min_j \{disp_j \text{ such that } type_j = \text{lb\_marker}\} & \text{otherwise} \end{cases}$$

Similarly, the **upper bound** of *Typemap* is defined to be

$$ub(Typemap) = \begin{cases} \max_j (disp_j + sizeof(type_j)) + \epsilon & \text{if no entry has type ub\_marker} \\ \max_j \{disp_j \text{ such that } type_j = \text{ub\_marker}\} & \text{otherwise} \end{cases}$$

Then

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

If $type_i$ requires alignment to a byte address that is a multiple of $k_i$, then $\epsilon$ is the least non-negative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$. In Fortran, it is implementation dependent whether the MPI implementation computes the alignments $k_i$ according to the alignments used by the compiler in common blocks, SEQUENCE derived types, BIND(C) derived types, or derived types that are neither SEQUENCE nor BIND(C).

The formal definitions given for the various datatype constructors apply now, with the amended definition of **extent**.

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

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

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

Example: Instead of

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

one should define

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

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

In principle, a compiler may define an additional alignment rule for structures, e.g., to use at least 4 or 8 byte alignment, although the content may have a $max_i k_i$ alignment less than this structure alignment. To maintain portability, users should always resize structure derived datatype handles if used in an array of structures, see the Example in Section 17.1.15. (*End of advice to users.*)

### 4.1.7 Extent and Bounds of Datatypes

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) |

**C binding**

```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
             MPI_Aint *extent)
```

**F08 binding**

```
MPI_Type_get_extent(datatype, lb, extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  lb, extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

MPI_TYPE_GET_EXTENT_X(datatype, lb, extent)

| | | |
|---|---|---|
| IN | datatype | datatype to get information on (handle) |
| OUT | lb | lower bound of datatype (integer) |
| OUT | extent | extent of datatype (integer) |

**C binding**

```
int MPI_Type_get_extent_x(MPI_Datatype datatype, MPI_Count *lb,
             MPI_Count *extent)
```

**F08 binding**

```
MPI_Type_get_extent_x(datatype, lb, extent, ierror)
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::  lb, extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_GET_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) LB, EXTENT
```

Returns the lower bound and the extent of datatype (as defined in Equation 4.1).

For both functions, if either OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

MPI allows one to change the extent of a datatype, using lower bound and upper bound markers. This provides control over the stride of successive datatypes that are replicated by datatype constructors, or are replicated by the count argument in a send or receive call.

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) |

**C binding**

```
int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
             MPI_Aint extent, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  lb, extent
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) LB, EXTENT
```

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 extent 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.

### 4.1.8 True Extent of Datatypes

Suppose we implement gather (see also Section 5.5) as a spanning tree implemented on top of point-to-point routines. Since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent, for example by using MPI_TYPE_CREATE_RESIZED. The functions MPI_TYPE_GET_TRUE_EXTENT and MPI_TYPE_GET_TRUE_EXTENT_X are provided which return the true extent of the datatype.

MPI_TYPE_GET_TRUE_EXTENT(datatype, true_lb, true_extent)

| | | |
|---|---|---|
| IN | datatype | datatype to get information on (handle) |
| OUT | true_lb | true lower bound of datatype (integer) |
| OUT | true_extent | true size of datatype (integer) |

**C binding**

```
int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
             MPI_Aint *true_extent)
```

**F08 binding**

```
MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: true_lb, true_extent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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_GET_TRUE_EXTENT_X(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) |

**C binding**

```
int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
             MPI_Count *true_extent)
```

**F08 binding**

```
MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) :: true_lb, true_extent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) TRUE_LB, TRUE_EXTENT
```

true_lb returns the offset of the lowest unit of store which is addressed by the datatype, i.e., the lower bound of the corresponding typemap, ignoring explicit lower bound markers. true_extent returns the true size of the datatype, i.e., the extent of the corresponding typemap, ignoring explicit lower bound and upper bound 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

$$true\_lb(Typemap) = min_j\{disp_j \ : \ type_j \neq \mathsf{lb\_marker}, \mathsf{ub\_marker}\},$$

$$true\_ub(Typemap) = max_j\{disp_j + sizeof(type_j) \ : \ type_j \neq \mathsf{lb\_marker}, \mathsf{ub\_marker}\},$$

and

$$true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$$

(Readers should compare this with the definitions in Section 4.1.6 and Section 4.1.7, 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.

For both functions, if either OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

### 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.”

MPI_TYPE_COMMIT(datatype)

| | | |
|---|---|---|
| INOUT | datatype | datatype that is committed (handle) |

**C binding**

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

**F08 binding**

```
MPI_Type_commit(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

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.

> *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. (*End of advice to implementors.*)

MPI_TYPE_COMMIT will accept a committed datatype; in this case, it is equivalent to a no-op.

**Example 4.10** The following code fragment gives examples of using MPI_TYPE_COMMIT.

```
INTEGER type1, type2
CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
              ! new type object created
CALL MPI_TYPE_COMMIT(type1, ierr)
              ! now type1 can be used for communication
type2 = type1
              ! type2 can be used for communication
              ! (it is a handle to same object as type1)
CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
              ! new uncommitted type object created
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) |

**C binding**

```
int MPI_Type_free(MPI_Datatype *datatype)
```

**F08 binding**

```
MPI_Type_free(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

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 then 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(oldtype, newtype)

| | | |
|---|---|---|
| IN | oldtype | datatype (handle) |
| OUT | newtype | copy of oldtype (handle) |

**C binding**

```
int MPI_Type_dup(MPI_Datatype oldtype, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_dup(oldtype, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_DUP(OLDTYPE, NEWTYPE, IERROR)
    INTEGER OLDTYPE, NEWTYPE, IERROR
```

MPI_TYPE_DUP is a type constructor which duplicates the existing oldtype 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 oldtype and any copied cached information, see Section 6.7.4. The new datatype has identical upper bound and lower bound and yields the same net result when fully decoded with the functions in Section 4.1.13. The newtype has the same committed state as the old oldtype.

### 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)
MPI_TYPE_FREE(newtype).
```

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

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

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

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

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

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

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

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

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

**Example 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,

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

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 $0 \le k \le \mathsf{count} \cdot n$. The number of basic elements received can be retrieved from status using the query functions MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X.

MPI_TYPE_GET_ELEMENTS(status, datatype, count)

| | | |
|---|---|---|
| IN | status | return status of receive operation (Status) |
| IN | datatype | datatype used by receive operation (handle) |
| OUT | count | number of received basic elements (non-negative integer) |

**C binding**

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

**F08 binding**

```
MPI_Type_get_elements(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_GET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

MPI_TYPE_GET_ELEMENTS_X(status, datatype, 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) |

**C binding**

```
int MPI_Type_get_elements_x(MPI_Status *status, MPI_Datatype datatype,
             MPI_Count *count)
```

**F08 binding**

```
MPI_Type_get_elements_x(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) COUNT
```

The datatype argument should match the argument provided by the receive call that set the status variable. For both functions, if the OUT parameter cannot express the value to be returned (e.g., if the parameter is too small to hold the output value), it is set to MPI_UNDEFINED.

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 $0 \leq k \leq$ count. If MPI_GET_COUNT returns $k$, then the number of basic elements received (and the value returned by MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X) 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 sets the value of count to MPI_UNDEFINED.

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

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

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

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

### 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.

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.

MPI_TYPE_GET_ENVELOPE(datatype, num_integers, num_addresses, num_datatypes, combiner)

| | | |
|---|---|---|
| IN | datatype | datatype to access (handle) |
| OUT | num_integers | number of input integers used in call constructing combiner (integer) |
| OUT | num_addresses | number of input addresses used in call constructing combiner (integer) |
| OUT | num_datatypes | number of input datatypes used in call constructing combiner (integer) |
| OUT | combiner | combiner (state) |

**C binding**

```
int MPI_Type_get_envelope(MPI_Datatype datatype, int *num_integers,
             int *num_addresses, int *num_datatypes, int *combiner)
```

**F08 binding**

```
MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
             combiner, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(OUT) :: num_integers, num_addresses, num_datatypes,
    combiner
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_GET_ENVELOPE(DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES,
             COMBINER, IERROR)
    INTEGER DATATYPE, NUM_INTEGERS, NUM_ADDRESSES, NUM_DATATYPES, COMBINER,
    IERROR
```

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. This is the most useful information and was felt to be reasonable even though it constrains implementations to remember the original constructor sequence even if the internal representation is different.

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

The list in Table 4.1 has the values that can be returned in combiner on the left and the call associated with them on the right.

| | |
|---|---|
| MPI_COMBINER_NAMED | a named predefined datatype |
| MPI_COMBINER_DUP | MPI_TYPE_DUP |
| MPI_COMBINER_CONTIGUOUS | MPI_TYPE_CONTIGUOUS |
| MPI_COMBINER_VECTOR | MPI_TYPE_VECTOR |
| MPI_COMBINER_HVECTOR | MPI_TYPE_CREATE_HVECTOR |
| MPI_COMBINER_INDEXED | MPI_TYPE_INDEXED |
| MPI_COMBINER_HINDEXED | MPI_TYPE_CREATE_HINDEXED |
| MPI_COMBINER_INDEXED_BLOCK | MPI_TYPE_CREATE_INDEXED_BLOCK |
| MPI_COMBINER_HINDEXED_BLOCK | MPI_TYPE_CREATE_HINDEXED_BLOCK |
| MPI_COMBINER_STRUCT | MPI_TYPE_CREATE_STRUCT |
| MPI_COMBINER_SUBARRAY | MPI_TYPE_CREATE_SUBARRAY |
| MPI_COMBINER_DARRAY | MPI_TYPE_CREATE_DARRAY |
| MPI_COMBINER_F90_REAL | MPI_TYPE_CREATE_F90_REAL |
| MPI_COMBINER_F90_COMPLEX | MPI_TYPE_CREATE_F90_COMPLEX |
| MPI_COMBINER_F90_INTEGER | MPI_TYPE_CREATE_F90_INTEGER |
| MPI_COMBINER_RESIZED | MPI_TYPE_CREATE_RESIZED |

Table 4.1: combiner values returned from MPI_TYPE_GET_ENVELOPE

If combiner is MPI_COMBINER_NAMED then datatype is a named predefined datatype.

The actual arguments used in the creation call for a datatype can be obtained using MPI_TYPE_GET_CONTENTS.

MPI_TYPE_GET_CONTENTS(datatype, max_integers, max_addresses, max_datatypes, array_of_integers, array_of_addresses, array_of_datatypes)

| | | |
|---|---|---|
| IN | datatype | datatype to access (handle) |
| IN | max_integers | number of elements in array_of_integers (integer) |
| IN | max_addresses | number of elements in array_of_addresses (integer) |
| IN | max_datatypes | number of elements in array_of_datatypes (integer) |
| OUT | array_of_integers | contains integer arguments used in constructing datatype (array of integers) |
| OUT | array_of_addresses | contains address arguments used in constructing datatype (array of integers) |
| OUT | array_of_datatypes | contains datatype arguments used in constructing datatype (array of handles) |

**C binding**

```
int MPI_Type_get_contents(MPI_Datatype datatype, int max_integers,
             int max_addresses, int max_datatypes, int array_of_integers[],
             MPI_Aint array_of_addresses[],
             MPI_Datatype array_of_datatypes[])
```

**F08 binding**

```
MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
             array_of_integers, array_of_addresses, array_of_datatypes,
             ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  max_integers, max_addresses, max_datatypes
    INTEGER, INTENT(OUT) ::  array_of_integers(max_integers)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
    array_of_addresses(max_addresses)
    TYPE(MPI_Datatype), INTENT(OUT) ::  array_of_datatypes(max_datatypes)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)
```

datatype must be a predefined unnamed or a derived datatype; the call is erroneous if datatype is a predefined named datatype.

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:

```
      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, IERROR)
      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];
MPI_Datatype type, d[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\n",
            LARGE);
    MPI_Abort(MPI_COMM_WORLD, 99);
};
MPI_Type_get_contents(type, ni, na, nd, i, a, d);
```

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 | Fortran location |
|---|---|---|
| oldtype | d[0] | D(1) |

and ni = 0, na = 0, nd = 1.

If combiner is MPI_COMBINER_CONTIGUOUS then

| Constructor argument | C | 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 | Fortran location |
|---|---|---|
| count | i[0] | I(1) |
| blocklength | i[1] | I(2) |
| stride | i[2] | I(3) |
| oldtype | d[0] | D(1) |

and ni = 3, na = 0, nd = 1.

If combiner is MPI_COMBINER_HVECTOR then

| Constructor argument | C | Fortran location |
|---|---|---|
| count | i[0] | I(1) |
| blocklength | i[1] | I(2) |
| stride | a[0] | A(1) |
| oldtype | d[0] | D(1) |

and ni = 2, na = 1, nd = 1.

If combiner is MPI_COMBINER_INDEXED then

| Constructor argument | C | Fortran location |
|---|---|---|
| count | i[0] | I(1) |
| array_of_blocklengths | i[1] to i[i[0]] | I(2) to I(I(1)+1) |
| array_of_displacements | i[i[0]+1] to i[2*i[0]] | I(I(1)+2) to I(2*I(1)+1) |
| oldtype | d[0] | D(1) |

and ni = 2*count+1, na = 0, nd = 1.

If combiner is MPI_COMBINER_HINDEXED then

| Constructor argument | C | Fortran location |
|---|---|---|
| count | i[0] | I(1) |
| array_of_blocklengths | i[1] to i[i[0]] | I(2) to I(I(1)+1) |
| array_of_displacements | a[0] to a[i[0]-1] | A(1) to A(I(1)) |
| oldtype | d[0] | D(1) |

and ni = count+1, na = count, nd = 1.

If combiner is MPI_COMBINER_INDEXED_BLOCK then

| Constructor argument | C | Fortran location |
|---|---|---|
| count | i[0] | I(1) |
| blocklength | i[1] | I(2) |
| array_of_displacements | i[2] to i[i[0]+1] | I(3) to I(I(1)+2) |
| oldtype | d[0] | D(1) |

and ni = count+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_HINDEXED_BLOCK then

| Constructor argument | C | Fortran location |
|---|---|---|
| count | i[0] | I(1) |
| blocklength | i[1] | I(2) |
| array_of_displacements | a[0] to a[i[0]-1] | A(1) to A(I(1)) |
| oldtype | d[0] | D(1) |

and ni = 2, na = count, nd = 1.

If combiner is MPI_COMBINER_STRUCT then

| Constructor argument | C | Fortran location |
|---|---|---|
| count | i[0] | I(1) |
| array_of_blocklengths | i[1] to i[i[0]] | I(2) to I(I(1)+1) |
| array_of_displacements | a[0] to a[i[0]-1] | A(1) to A(I(1)) |
| array_of_types | d[0] to d[i[0]-1] | D(1) to D(I(1)) |

and ni = count+1, na = count, nd = count.

If combiner is MPI_COMBINER_SUBARRAY then

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

and ni = 3*ndims+2, na = 0, nd = 1.

If combiner is MPI_COMBINER_DARRAY then

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

and ni = 4*ndims+4, na = 0, nd = 1.

If combiner is MPI_COMBINER_F90_REAL then

| Constructor argument | C | Fortran location |
|---|---|---|
| p | i[0] | I(1) |
| r | i[1] | I(2) |

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_COMPLEX then

| Constructor argument | C | Fortran location |
|---|---|---|
| p | i[0] | I(1) |
| r | i[1] | I(2) |

and ni = 2, na = 0, nd = 0.

If combiner is MPI_COMBINER_F90_INTEGER then

| Constructor argument | C | Fortran location |
|---|---|---|
| r | i[0] | I(1) |

and ni = 1, na = 0, nd = 0.

If combiner is MPI_COMBINER_RESIZED then

| Constructor argument | C | Fortran location |
|---|---|---|
| lb | a[0] | A(1) |
| extent | a[1] | A(2) |
| oldtype | d[0] | D(1) |

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, myrank, ierr
      INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
      INTEGER status(MPI_STATUS_SIZE)

C     extract the section a(1:17:2, 3:11, 2:10)
C     and store it in e(:,:,:).

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)

C     create datatype for a 1D section
      CALL MPI_TYPE_VECTOR(9, 1, 2, MPI_REAL, oneslice, ierr)

C     create datatype for a 2D section
      CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*sizeofreal, oneslice,
                                   twoslice, ierr)

C     create datatype for the entire section
      CALL MPI_TYPE_CREATE_HVECTOR(9, 1, 100*100*sizeofreal, twoslice,
                                   threeslice, ierr)

      CALL MPI_TYPE_COMMIT(threeslice, ierr)
      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.

```
      REAL a(100,100), b(100,100)
      INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
      INTEGER status(MPI_STATUS_SIZE)

C     copy lower triangular part of array a
C     onto lower triangular part of array b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

C     compute start and size of each column
      DO i=1, 100
        disp(i) = 100*(i-1) + i
        blocklen(i) = 100-i
      END DO

C     create datatype for lower triangular part
      CALL MPI_TYPE_INDEXED(100, blocklen, disp, MPI_REAL, ltype, ierr)

      CALL MPI_TYPE_COMMIT(ltype, ierr)
      CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1,
                  ltype, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

**Example 4.15** Transpose a matrix.

```
      REAL a(100,100), b(100,100)
      INTEGER row, xpose, myrank, ierr
      INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
      INTEGER status(MPI_STATUS_SIZE)

C     transpose matrix a onto b

      CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)

      CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, 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_CREATE_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 row, row1
      INTEGER (KIND=MPI_ADDRESS_KIND) disp(2), lb, 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_GET_EXTENT(MPI_REAL, lb, 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
      lb = 0
      CALL MPI_TYPE_CREATE_RESIZED(row, lb, sizeofreal, row1, ierr)

      CALL MPI_TYPE_COMMIT(row1, ierr)

C     send 100 rows and receive in column major order
      CALL MPI_SENDRECV(a, 100, row1, myrank, 0, b, 100*100,
                        MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

**Example 4.17** We manipulate an array of structures.

```
struct Partstruct
{
   int    type;  /* particle type */
   double d[6];   /* particle coordinates */
   char   b[7];   /* some additional information */
};

struct Partstruct    particle[1000];

int          i, dest, tag;
MPI_Comm     comm;


/* build datatype describing structure */

MPI_Datatype Particlestruct, Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
MPI_Aint     disp[3];
MPI_Aint     base, lb, sizeofentry;
```

```
/* compute displacements of structure components */

MPI_Get_address(particle, disp);
MPI_Get_address(particle[0].d, disp+1);
MPI_Get_address(particle[0].b, disp+2);
base = disp[0];
for (i=0; i < 3; i++) disp[i] = MPI_Aint_diff(disp[i], base);

MPI_Type_create_struct(3, blocklen, disp, type, &Particlestruct);

   /* If compiler does padding in mysterious ways,
   the following may be safer */

/* compute extent of the structure */

MPI_Get_address(particle+1, &sizeofentry);
sizeofentry = MPI_Aint_diff(sizeofentry, base);

/* build datatype describing structure */

MPI_Type_create_resized(Particlestruct, 0, sizeofentry, &Particletype);


              /* 4.1:
        send the entire array */

MPI_Type_commit(&Particletype);
MPI_Send(particle, 1000, Particletype, dest, tag, comm);


              /* 4.2:
        send only the entries of type zero particles,
        preceded by the number of such entries */

MPI_Datatype Zparticles;   /* datatype describing all particles
                              with type zero (needs to be recomputed
                              if types change) */
MPI_Datatype Ztype;

int          zdisp[1000];
int          zblock[1000], j, k;
int          zzblock[2] = {1,1};
MPI_Aint     zzdisp[2];
MPI_Datatype zztype[2];

/* compute displacements of type zero particles */
j = 0;
```

```
for (i=0; i < 1000; i++)
   if (particle[i].type == 0)
      {
        zdisp[j] = i;
        zblock[j] = 1;
        j++;
      }

/* create datatype for type zero particles  */
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);

/* prepend particle count */
MPI_Get_address(&j, zzdisp);
MPI_Get_address(particle, zzdisp+1);
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_create_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].type == 0)
      {
         for (k=i+1; (k < 1000)&&(particle[k].type == 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_Type_get_extent(Particletype, &lb, &sizeofentry);

     /* sizeofentry can also be computed by subtracting the address
        of particle[0] from the address of particle[1] */
```

```
MPI_Type_create_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 Twodouble;

MPI_Type_contiguous(2, MPI_DOUBLE, &Twodouble);

MPI_Datatype Onepair;   /* datatype for one pair of coordinates, with
                          the extent of one particle entry */

MPI_Type_create_resized(Twodouble, 0, sizeofentry, &Onepair );
MPI_Type_commit(&Onepair);
MPI_Send(particle[0].d, 1000, Onepair, dest, tag, comm);
```

**Example 4.18** The same manipulations as in the previous example, but use absolute addresses in datatypes.

```
struct Partstruct
{
    int    type;
    double d[6];
    char   b[7];
};

struct Partstruct particle[1000];

           /* build datatype describing first array entry */

MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          block[3] = {1, 6, 7};
MPI_Aint     disp[3];

MPI_Get_address(particle, disp);
MPI_Get_address(particle[0].d, disp+1);
MPI_Get_address(particle[0].b, disp+2);
MPI_Type_create_struct(3, block, disp, type, &Particletype);

/* Particletype describes first array entry -- using absolute
   addresses */

                  /* 5.1:
            send the entire array */
```

```
MPI_Type_commit(&Particletype);
MPI_Send(MPI_BOTTOM, 1000, Particletype, dest, tag, comm);


                 /* 5.2:
        send the entries of type zero,
        preceded by the number of such entries */

MPI_Datatype Zparticles, Ztype;

int          zdisp[1000];
int          zblock[1000], i, j, k;
int          zzblock[2] = {1,1};
MPI_Datatype zztype[2];
MPI_Aint     zzdisp[2];

j=0;
for (i=0; i < 1000; i++)
    if (particle[i].type == 0)
        {
            for (k=i+1; (k < 1000)&&(particle[k].type == 0); k++);
            zdisp[j] = i;
            zblock[j] = k-i;
            j++;
            i = k;
        }
MPI_Type_indexed(j, zblock, zdisp, Particletype, &Zparticles);
/* Zparticles describe particles with type zero, using
   their absolute addresses*/

/* prepend particle count */
MPI_Get_address(&j, zzdisp);
zzdisp[1] = (MPI_Aint)0;
zztype[0] = MPI_INT;
zztype[1] = Zparticles;
MPI_Type_create_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   mpi_utype[2];
MPI_Aint       i, extent;

/* compute an MPI datatype for each possible union type;
   assume values are left-aligned in union storage. */

MPI_Get_address(u, &i);
MPI_Get_address(u+1, &extent);
extent = MPI_Aint_diff(extent, i);

MPI_Type_create_resized(MPI_INT, 0, extent, &mpi_utype[0]);

MPI_Type_create_resized(MPI_FLOAT, 0, extent, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);

/* actual communication */

MPI_Send(u, 1000, mpi_utype[utype], dest, tag, comm);
```

**Example 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.

```
/*
  Example of decoding a datatype.

  Returns 0 if the datatype is predefined, 1 otherwise
 */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int printdatatype(MPI_Datatype datatype)
{
    int *array_of_ints;
    MPI_Aint *array_of_adds;
    MPI_Datatype *array_of_dtypes;
    int num_ints, num_adds, num_dtypes, combiner;
    int i;

    MPI_Type_get_envelope(datatype,
```

```
                        &num_ints, &num_adds, &num_dtypes, &combiner);
    switch (combiner) {
    case MPI_COMBINER_NAMED:
        printf("Datatype is named:");
        /* To print the specific type, we can match against the
           predefined forms. We can NOT use a switch statement here
           We could also use MPI_TYPE_GET_NAME if we prefered to use
           names that the user may have changed.
         */
        if      (datatype == MPI_INT)    printf("MPI_INT\n");
        else if (datatype == MPI_DOUBLE) printf("MPI_DOUBLE\n");
        ... else test for other types ...
        return 0;
        break;
    case MPI_COMBINER_STRUCT:
    case MPI_COMBINER_STRUCT_INTEGER:
        printf("Datatype is struct containing");
        array_of_ints   = (int *)malloc(num_ints * sizeof(int));
        array_of_adds   =
                   (MPI_Aint *) malloc(num_adds * sizeof(MPI_Aint));
        array_of_dtypes = (MPI_Datatype *)
            malloc(num_dtypes * sizeof(MPI_Datatype));
        MPI_Type_get_contents(datatype, num_ints, num_adds, num_dtypes,
                           array_of_ints, array_of_adds, array_of_dtypes);
        printf(" %d datatypes:\n", array_of_ints[0]);
        for (i=0; i<array_of_ints[0]; i++) {
            printf("blocklength %d, displacement %ld, type:\n",
                    array_of_ints[i+1], (long)array_of_adds[i]);
            if (printdatatype(array_of_dtypes[i])) {
                /* Note that we free the type ONLY if it
                   is not predefined */
                MPI_Type_free(&array_of_dtypes[i]);
            }
        }
        free(array_of_ints);
        free(array_of_adds);
        free(array_of_dtypes);
        break;
        ... other combiner values ...
    default:
        printf("Unrecognized combiner type\n");
    }
    return 1;
}
```

## 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.

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) |

**C binding**

```
int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outsize, int *position, MPI_Comm comm)
```

**F08 binding**

```
MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    INTEGER, INTENT(IN) ::  incount, outsize
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(*), DIMENSION(..)  ::  outbuf
    INTEGER, INTENT(INOUT) ::  position
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

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 (non-negative integer) |
| IN | datatype | datatype of each output data item (handle) |
| IN | comm | communicator for packed message (handle) |

**C binding**

```
int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,
             int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    INTEGER, INTENT(IN) ::  insize, outcount
    INTEGER, INTENT(INOUT) ::  position
    TYPE(*), DIMENSION(..)  ::  outbuf
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR
```

Unpacks a message into the receive buffer specified by outbuf, outcount, datatype from the buffer space specified by inbuf and insize. The output buffer can be any communication buffer allowed in MPI_RECV. The input buffer is a contiguous storage area containing insize bytes, starting at address inbuf. The input value of position is the first location in the input buffer occupied by the packed message. position is incremented by the size of the packed message, so that the output value of position is the first location in the input buffer after the locations occupied by the message that was unpacked. comm is the communicator used to receive the packed message.

*Advice to users.* Note the difference between MPI_RECV and MPI_UNPACK: in MPI_RECV, the count argument specifies the maximum number of items that can be received. The actual number of items received is determined by the length of the incoming message. In MPI_UNPACK, the count argument specifies the actual number of items that are unpacked; the "size" of the corresponding message is the increment in position. The reason for this change is that the "incoming message size" is not predetermined since the user decides how much to unpack; nor is it easy to determine the "message size" from the number of items to be unpacked. In fact, in a heterogeneous system, this number may not be determined *a priori*. (*End of advice to users.*)

To understand the behavior of pack and unpack, it is convenient to think of the data part of a message as being the sequence obtained by concatenating the successive values sent in that message. The pack operation stores this sequence in the buffer space, as if sending the message to that buffer. The unpack operation retrieves this sequence from buffer space, as if receiving a message from that buffer. (It is helpful to think of internal Fortran files or `sscanf` in C, for a similar function.)

Several messages can be successively packed into one **packing unit**. This is effected by several successive **related** calls to MPI_PACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for outbuf, outcount and comm. This packing unit now contains the equivalent information that would have been stored in a message by one send call with a send buffer that is the "concatenation" of the individual send buffers.

A packing unit can be sent using type MPI_PACKED. Any point to point or collective communication function can be used to move the sequence of bytes that forms the packing unit from one process to another. This packing unit can now be received using any receive operation, with any datatype: the type matching rules are relaxed for messages sent with type MPI_PACKED.

A message sent with any type (including MPI_PACKED) can be received using the type MPI_PACKED. Such a message can then be unpacked by calls to MPI_UNPACK.

A packing unit (or a message created by a regular, "typed" send) can be unpacked into several successive messages. This is effected by several successive related calls to MPI_UNPACK, where the first call provides position = 0, and each successive call inputs the value of position that was output by the previous call, and the same values for inbuf, insize and comm.

The concatenation of two packing units is not necessarily a packing unit; nor is a substring 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 | incount | 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) |

**C binding**

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
             int *size)
```

**F08 binding**

```
MPI_Pack_size(incount, datatype, comm, size, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
    INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR
```

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). If the packed size of the datatype cannot be expressed by the size parameter, then MPI_PACK_SIZE sets the value of size to MPI_UNDEFINED.

> *Rationale.* The call returns an upper bound, rather than an exact bound, since the exact amount of space needed to pack the message may depend on the context (e.g., first message packed in a packing unit may take more space). (*End of rationale.*)

**Example 4.21** An example using MPI_PACK.

```
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, MPI_STATUS_IGNORE);
```

**Example 4.22** An elaborate example.

```
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;
    MPI_Get_address(&i, disp);
    MPI_Get_address(a, disp+1);
    type[0] = MPI_INT;
    type[1] = MPI_FLOAT;
    MPI_Type_create_struct(2, len, disp, type, &newtype);
    MPI_Type_commit(&newtype);

    /* Pack i followed by a[0]...a[i-1]*/

    position = 0;
    MPI_Pack(MPI_BOTTOM, 1, newtype, buff, 1000, &position, MPI_COMM_WORLD);

    /* Send */

    MPI_Send(buff, position, MPI_PACKED, 1, 0,
             MPI_COMM_WORLD);

/* *****
   One can replace the last three lines with
   MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
   ***** */
}
else if (myrank == 1)
```

```
{
    /* RECEIVER CODE */

    MPI_Status status;

    /* Receive */

    MPI_Recv(buff, 1000, MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);

    /* Unpack i */

    position = 0;
    MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT, MPI_COMM_WORLD);

    /* Unpack a[0]...a[i-1] */
    MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT, MPI_COMM_WORLD);
}
```

**Example 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,
     displs[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, 0,
               MPI_DATATYPE_NULL, root, comm);

    /* gather at root packed messages */
    MPI_Gatherv(lbuf, position, MPI_PACKED, NULL,
                NULL, NULL, MPI_DATATYPE_NULL, root, comm);
```

```
} 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;
    }
    cbuf[concat_pos] = '\0';
}
```

## 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 (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 (integer) |
| INOUT | position | current position in buffer, in bytes (integer) |

**C binding**

```
int MPI_Pack_external(const char *datarep, const void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
             MPI_Aint *position)
```

**F08 binding**

```
MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
             position, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  datarep
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    INTEGER, INTENT(IN) ::  incount
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(*), DIMENSION(..)  ::  outbuf
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  outsize
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) ::  position
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
             POSITION, IERROR)
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION
```

MPI_UNPACK_EXTERNAL(datarep, inbuf, insize, position, outbuf, outsize, datatype)

| | | |
|---|---|---|
| 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 | outsize | number of output data items (non-negative integer) |
| IN | datatype | datatype of output data item (handle) |

**C binding**

```
int MPI_Unpack_external(const char *datarep, const void *inbuf,
             MPI_Aint insize, MPI_Aint *position, void *outbuf,
             int outsize, MPI_Datatype datatype)
```

**F08 binding**

```
MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outsize,
             datatype, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  datarep
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  insize
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) ::  position
    TYPE(*), DIMENSION(..)  ::  outbuf
    INTEGER, INTENT(IN) ::  outsize
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTSIZE,
             DATATYPE, IERROR)
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
    INTEGER OUTSIZE, DATATYPE, IERROR
```

MPI_PACK_EXTERNAL_SIZE(datarep, incount, datatype, size)

| | | |
|---|---|---|
| IN | datarep | data representation (string) |
| IN | incount | number of input data items (non-negative integer) |
| IN | datatype | datatype of each input data item (handle) |
| OUT | size | output buffer size, in bytes (integer) |

**C binding**

```
int MPI_Pack_external_size(const char *datarep, int incount,
             MPI_Datatype datatype, MPI_Aint *size)
```

**F08 binding**

```
MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    CHARACTER*(*) DATAREP
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
```

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) |

# 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, MPI_IBARRIER: Barrier synchronization across all members of a group (Section 5.3 and Section 5.12.1).
- MPI_BCAST, MPI_IBCAST: Broadcast from one member to all members of a group (Section 5.4 and Section 5.12.2). This is shown as "broadcast" in Figure 5.1.
- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV: Gather data from all members of a group to one member (Section 5.5 and Section 5.12.3). This is shown as "gather" in Figure 5.1.
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV: Scatter data from one member to all members of a group (Section 5.6 and Section 5.12.4). This is shown as "scatter" in Figure 5.1.
- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV: A variation on Gather where all members of a group receive the result (Section 5.7 and Section 5.12.5). This is shown as "allgather" in Figure 5.1.
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW: Scatter/Gather data from all members to all members of a group (also called complete exchange) (Section 5.8 and Section 5.12.6). This is shown as "complete exchange" in Figure 5.1.
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE: Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all members of a group (Section 5.9.6 and Section 5.12.8) and a variation where the result is returned to only one member (Section 5.9 and Section 5.12.7).
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER: A combined reduction and scatter operation (Section 5.10, Section 5.12.9, and Section 5.12.10).

- MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN: Scan across all members of a group (also called prefix) (Section 5.11, Section 5.11.2, Section 5.12.11, and Section 5.12.12).

One of the key arguments in a call to a collective routine is a communicator that defines the group or groups of participating processes and provides a context for the operation. This is discussed further in Section 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 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 operations can (but are not required to) complete as soon as the caller's participation in the collective communication is finished. A blocking operation is complete as soon as the call returns. A nonblocking (immediate) call requires a separate completion call (cf. Section 3.7). The completion of a collective operation indicates that the caller is free to modify 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 the description of the operation). Thus, a collective communication operation may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier operation.

Collective communication calls may use the same communicators as point-to-point communication; MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication. The collective operations do not have a message tag argument. A more detailed discussion of correct use of collective routines is found in Section 5.14.

*Rationale.* The equal-data restriction (on type matching) was made so as to avoid the complexity of providing a facility analogous to the status argument of MPI_RECV for discovering the amount of data sent. Some of the collective routines would require an array of status values.

The statements about synchronization are made so as to allow a variety of implementations of the collective functions.

(*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.

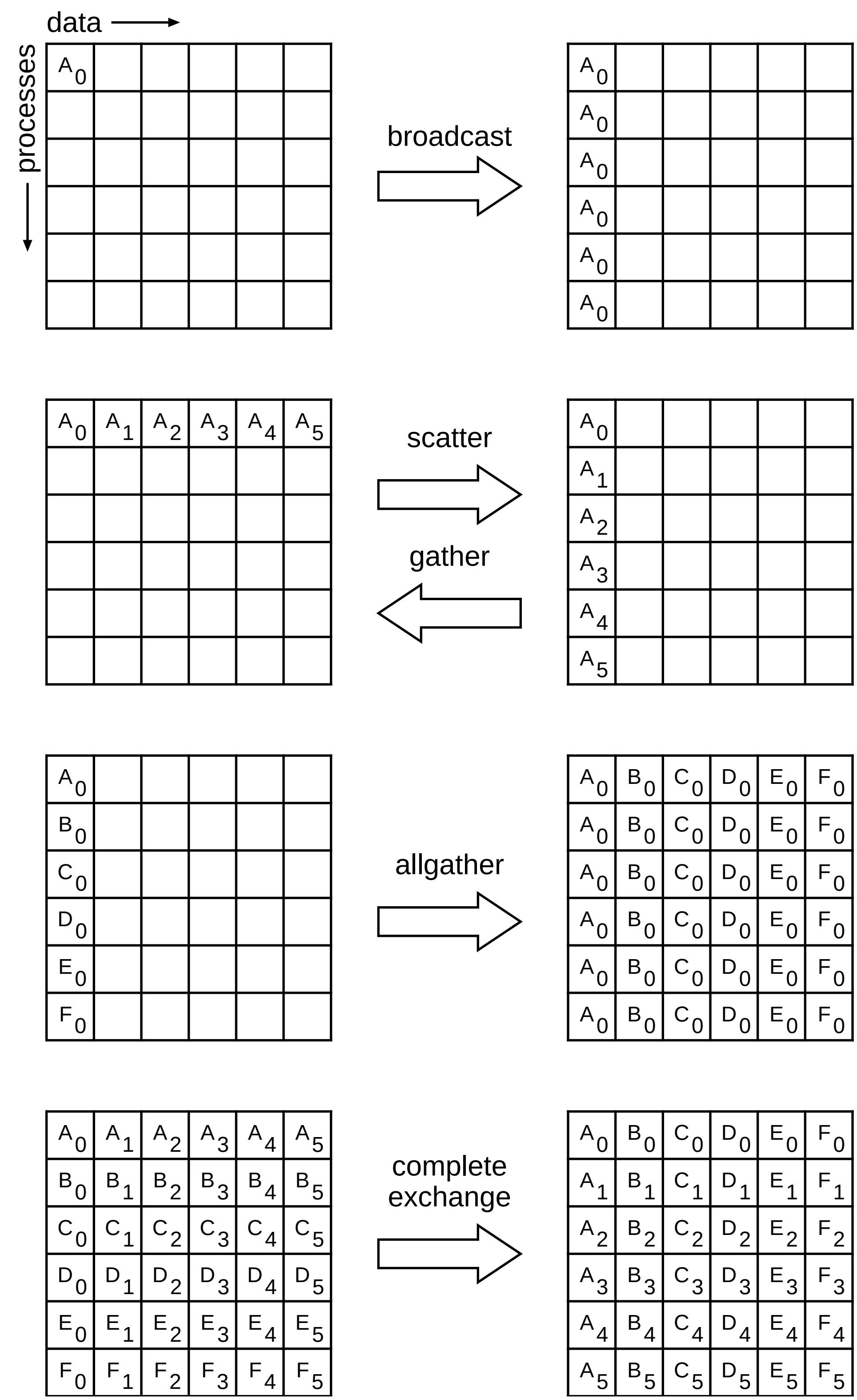


Figure 5.1: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data $A_0$, but after the broadcast all processes contain it.

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

*Advice to implementors.* While vendors may write optimized collective routines matched to their architectures, a complete library of the collective communication routines can be written entirely using the MPI point-to-point communication functions and a few auxiliary functions. If implementing on top of point-to-point, a hidden, special communicator might be created for the collective operation so as to avoid interference with any on-going point-to-point communication at the time of the collective call. This is discussed further in Section 5.14. (*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.

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. (*End of advice to users.*)

### 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, [55]):

**All-To-All** All processes contribute to the result. All processes receive the result.

- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER
- MPI_BARRIER, MPI_IBARRIER

**All-To-One** All processes contribute to the result. One process receives the result.

- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV
- MPI_REDUCE, MPI_IREDUCE

**One-To-All** One process contributes to the result. All processes receive the result.

- MPI_BCAST, MPI_IBCAST
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV

**Other** Collective operations that do not fit into one of the above categories.

- MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, MPI_IEXSCAN

The data movement patterns of MPI_SCAN, MPI_ISCAN, MPI_EXSCAN, and MPI_IEXSCAN do not fit this taxonomy.

The application of collective communication to 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_IBARRIER
- MPI_BCAST, MPI_IBCAST
- MPI_GATHER, MPI_IGATHER, MPI_GATHERV, MPI_IGATHERV,
- MPI_SCATTER, MPI_ISCATTER, MPI_SCATTERV, MPI_ISCATTERV,

- MPI_ALLGATHER, MPI_IALLGATHER, MPI_ALLGATHERV, MPI_IALLGATHERV,
- MPI_ALLTOALL, MPI_IALLTOALL, MPI_ALLTOALLV, MPI_IALLTOALLV, MPI_ALLTOALLW, MPI_IALLTOALLW,
- MPI_ALLREDUCE, MPI_IALLREDUCE, MPI_REDUCE, MPI_IREDUCE,
- MPI_REDUCE_SCATTER_BLOCK, MPI_IREDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_IREDUCE_SCATTER.

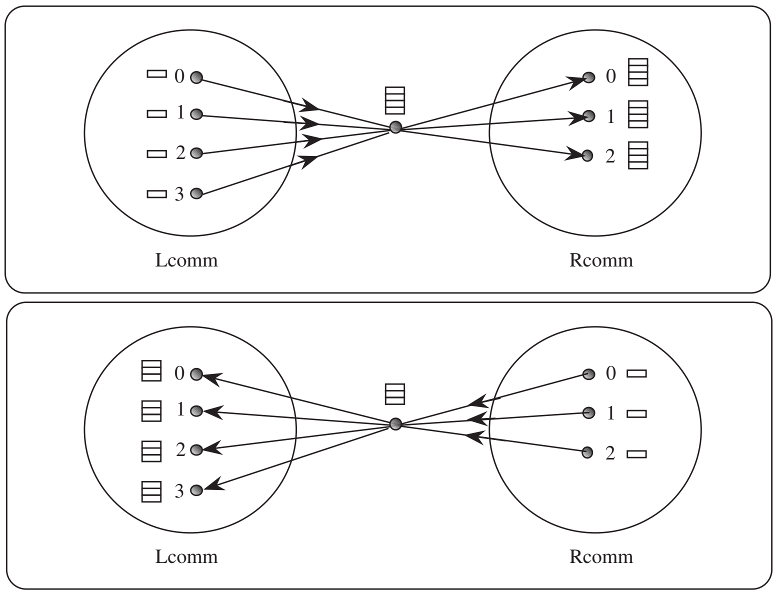


Figure 5.2: Intercommunicator allgather. The focus of data to one process is represented, not mandated by the semantics. The two phases do allgathers in both directions.

### 5.2.3 Specifics for Intercommunicator Collective Operations

All processes in both groups identified by the intercommunicator must call the collective routine.

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 in the All-To-One or One-To-All categories, 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 in the All-To-All category, then the transfer is bidirectional.

*Rationale.* Operations in the All-To-One and One-To-All categories are unidirectional by nature, and there is a clear way of specifying direction. Operations in the All-To-All

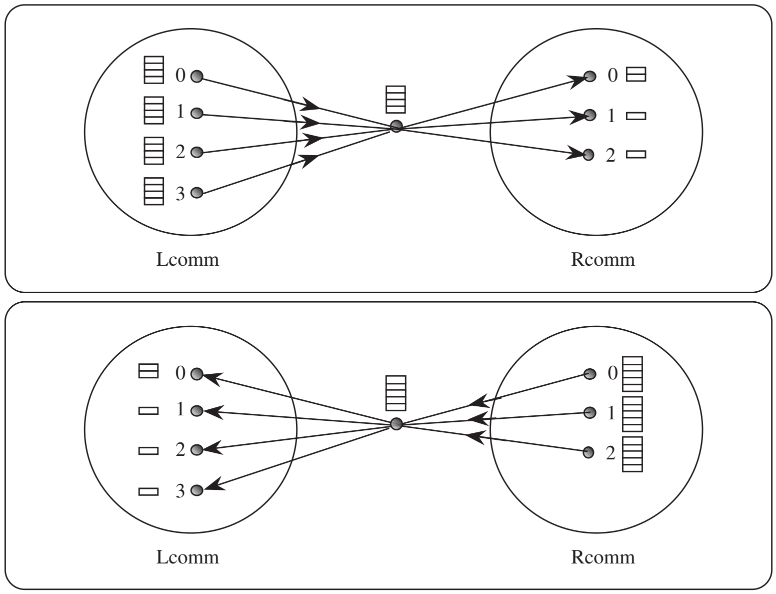


Figure 5.3: Intercommunicator reduce-scatter. The focus of data to one process is represented, not mandated by the semantics. The two phases do reduce-scatters in both directions.

category 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

MPI_BARRIER(comm)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |

**C binding**

```
int MPI_Barrier(MPI_Comm comm)
```

**F08 binding**

```
MPI_Barrier(comm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
```

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, MPI_BARRIER involves two groups. The call returns at processes in one group (group A) of the intercommunicator only after all members of the other group (group B) have entered the call (and vice versa). A process may return from the call before all processes in its own group have entered the call.

## 5.4 Broadcast

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) |

**C binding**

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm)
```

**F08 binding**

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  buffer
    INTEGER, INTENT(IN) ::  count, root
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

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_ROOT in root. All other processes in group A pass the value MPI_PROC_NULL in root. Data is broadcast from the root to all processes in group B. The buffer arguments of the processes in group B must be consistent with the buffer argument of the root.

### 5.4.1 Example using MPI_BCAST

The examples in this section use intracommunicators.

**Example 5.1**

Broadcast 100 `int`s from process `0` to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast(array, 100, MPI_INT, root, comm);
```

As in many of our example code fragments, we assume that some of the variables (such as `comm` in the above) have been assigned appropriate values.

## 5.5 Gather

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, 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, significant only at root) |
| IN | recvcount | number of elements for any single receive (non-negative integer, significant only at root) |
| IN | recvtype | data type of recv buffer elements (significant only at root) (handle) |
| IN | root | rank of receiving process (integer) |
| IN | comm | communicator (handle) |

**C binding**

```
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
             MPI_Comm comm)
```

**F08 binding**

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
```

```
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

If 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

MPI_Send(sendbuf, sendcount, sendtype, root , ...),

and the root had executed n calls to

MPI_Recv(recvbuf+i· recvcount· extent(recvtype), recvcount, recvtype, i,...),

where extent(recvtype) is the type extent obtained from a call to MPI_Type_get_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 MPI_RECV(recvbuf, recvcount·n, recvtype, ...).

The receive buffer is ignored for all non-root processes.

General, derived datatypes are allowed for both sendtype and recvtype. The type signature of sendcount, sendtype on each process must be equal to the type signature of recvcount, 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.

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 and types should not cause any location on the root to be written more than once. Such a call is erroneous.

Note that the 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 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.

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, 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, significant only at root) |
| IN | recvcounts | non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root) |
| IN | displs | integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root) |
| IN | recvtype | data type of recv buffer elements (significant only at root) (handle) |
| IN | root | rank of receiving process (integer) |
| IN | comm | communicator (handle) |

**C binding**

```
int MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, const int recvcounts[], const int displs[],
             MPI_Datatype recvtype, int root, MPI_Comm comm)
```

**F08 binding**

```
MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcounts(*), displs(*), root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
              COMM, IERROR
```

MPI_GATHERV extends the functionality of MPI_GATHER by allowing a varying count of data from each process, since recvcounts is now an array. It also allows more flexibility as to where the data is placed on the root, by providing the new argument, displs.

If comm is an intracommunicator, the outcome is *as if* each process, including the root process, sends a message to the root,

MPI_Send(sendbuf, sendcount, sendtype, root, ...),

and the root executes n receives,

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[i], 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 `int`s from every process in group to root. See Figure 5.4.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
...
MPI_Comm_size(comm, &gsize);
```

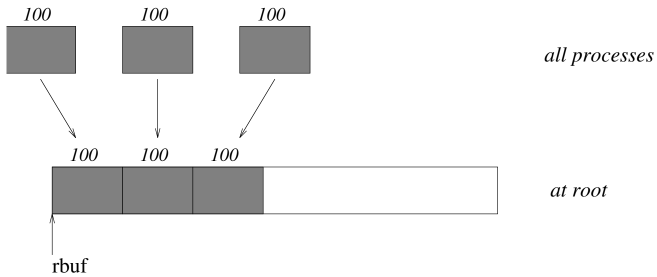


Figure 5.4: The root process gathers 100 `ints` from each process in the group.

```
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.

```
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.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
...
MPI_Comm_size(comm, &gsize);
MPI_Type_contiguous(100, MPI_INT, &rtype);
MPI_Type_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* $\geq 100$. See Figure 5.5.

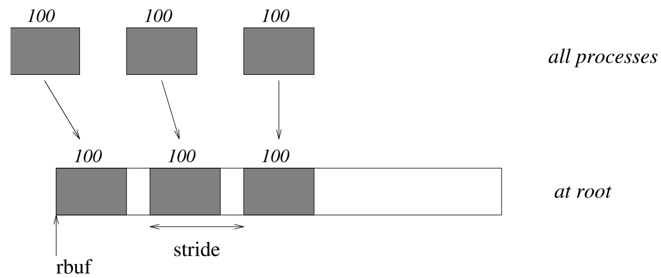


Figure 5.5: The root process gathers 100 `int`s from each process in the group, each set is placed `stride` `int`s apart.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;

...

MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv(sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
            root, comm);
```

Note that the program is erroneous if $stride < 100$.

**Example 5.6**

Same as Example 5.5 on the receiving side, but send the 100 `int`s from the 0th column of a 100×150 `int` array, in C. See Figure 5.6.

```
MPI_Comm comm;
int gsize,sendarray[100][150];
int root, *rbuf, stride;
MPI_Datatype stype;
int *displs,i,*rcounts;

...

MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
```

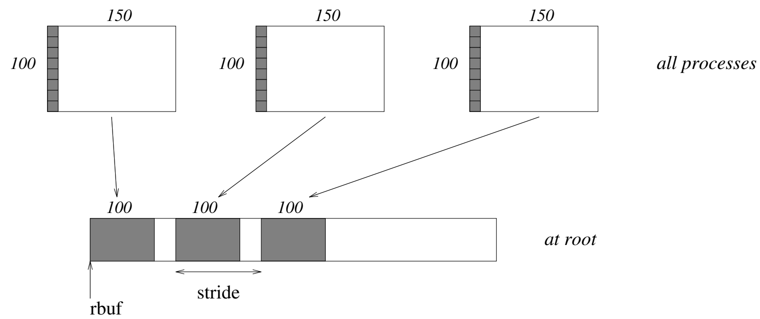


Figure 5.6: The root process gathers column `0` of a 100×150 C array, and each set is placed `stride ints` apart.

```
        displs[i] = i*stride;
        rcounts[i] = 100;
    }
    /* Create datatype for 1 column of array
     */
    MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
    MPI_Type_commit(&stype);
    MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT,
                root, comm);
```

**Example 5.7**

Process `i` sends `(100-i) ints` from the `i`-th column of a 100 × 150 `int` array, in C. It is received into a buffer with stride, as in the previous two examples. See Figure 5.7.

```
    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);
```

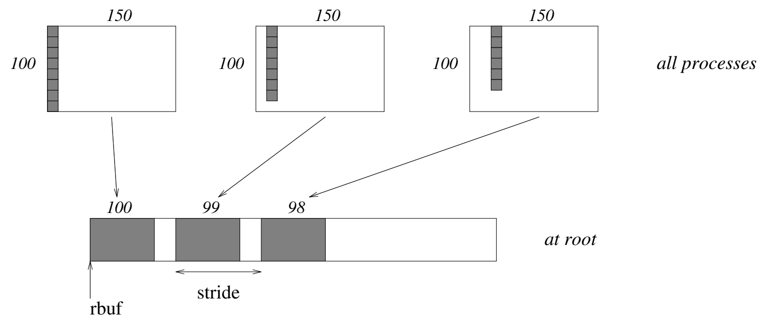


Figure 5.7: The root process gathers `100-i` `ints` from column `i` of a 100×150 C array, and each set is placed `stride` `ints` apart.

```
/* 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.

```
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;
}
/* Create datatype for one int, with extent of entire row
 */
MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &stype);
MPI_Type_commit(&stype);
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
```

```
            root, comm);
```

**Example 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.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs,i,*rcounts,offset;

...

MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 */

/* set up displs and rcounts vectors first
 */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
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**

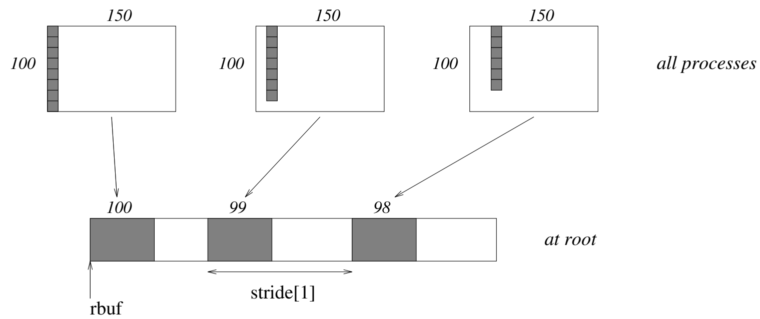


Figure 5.8: The root process gathers `100-i` `int`s from column `i` of a 100×150 C array, and each set is placed `stride[i]` `int`s apart (a varying stride).

Process `i` sends `num` `int`s from the `i`-th column of a $100 \times 150$ `int` array, in C. The complicating factor is that the various values of `num` are not known to `root`, so a separate gather must first be run to find these out. The data is placed contiguously at the receiving end.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, myrank;
MPI_Datatype stype;
int *displs,i,*rcounts,num;

...

MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

/* First, gather nums to root
 */
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
 */
MPI_Type_create_resized(MPI_INT, 0, 150*sizeof(int), &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(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

| | | |
|---|---|---|
| IN | sendbuf | address of send buffer (choice, significant only at root) |
| IN | sendcount | number of elements sent to each process (non-negative integer, significant only at root) |
| IN | sendtype | data type of send buffer elements (significant only at root) (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 binding**

```
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
             MPI_Comm comm)
```

**F08 binding**

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

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 a case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain $n$ segments, where $n$ is the group size; the *root*-th segment, which root should "send to itself," is not moved.

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.

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 rank |
| 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 binding**

```
int MPI_Scatterv(const void* sendbuf, const int sendcounts[],
             const int displs[], MPI_Datatype sendtype, void* recvbuf,
             int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

**F08 binding**

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
             recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
              COMM, IERROR
```

MPI_SCATTERV is the inverse operation to MPI_GATHERV.

MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process, since sendcounts is now an array. It also allows more flexibility as to where the data is taken from on the root, by providing an additional argument, displs.

If comm is an intracommunicator, the outcome is as if the root executed n send operations, MPI_Send(sendbuf+displs[i]· extent(sendtype), sendcounts[i], sendtype, i,...),

and each process executed a receive,

MPI_Recv(recvbuf, recvcount, recvtype, i,...).

The send buffer is ignored for all non-root processes.

The type signature implied by sendcount[i], sendtype at the root must be equal to the type signature implied by recvcount, recvtype at process i (however, the type maps may be different). This implies that the amount of data sent must be equal to the amount of data received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

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, types, and displacements should not cause any location on the root to be read more than once.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE as the value of recvbuf at the root. In such a case, recvcount and recvtype are ignored, and root "sends" no data to itself. The scattered vector is still assumed to contain $n$ segments, where $n$ is the group size; the *root*-th segment, which root should "send to itself," is not moved.

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.

### 5.6.1 Examples using MPI_SCATTER, MPI_SCATTERV

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.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

**Example 5.12**

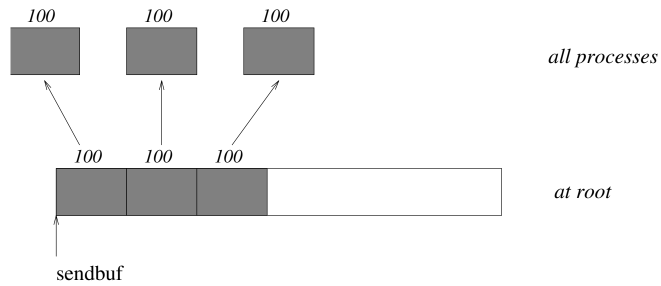


Figure 5.9: The root process scatters sets of 100 `int`s to each process in the group.

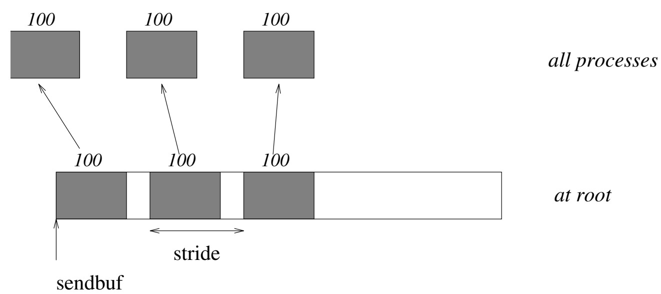


Figure 5.10: The root process scatters sets of 100 `int`s, moving by `stride ints` from send to send in the scatter.

The reverse of Example 5.5. The root process scatters sets of 100 `int`s to the other processes, but the sets of 100 are *stride ints* apart in the sending buffer. Requires use of MPI_SCATTERV. Assume *stride* $\geq$ 100. See Figure 5.10.

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100], i, *displs, *scounts;

...

MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
             root, comm);
```

**Example 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.

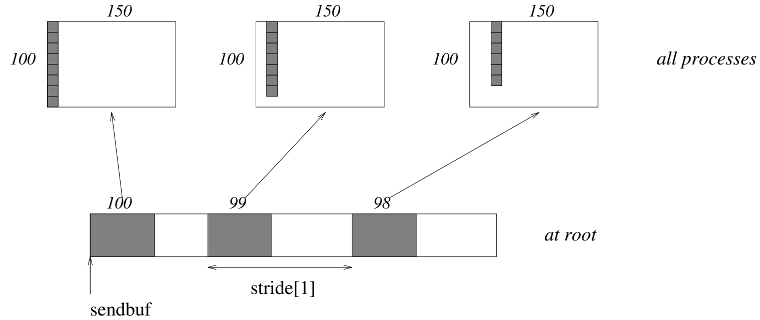


Figure 5.11: The root scatters blocks of `100-i` `int`s into column `i` of a 100×150 C array. At the sending side, the blocks are `stride[i]` `int`s apart.

```
MPI_Comm comm;
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
...
MPI_Comm_size(comm, &gsize);
MPI_Comm_rank(comm, &myrank);

stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow
 * sendbuf comes from elsewhere
 */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    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);
```

## 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) |

**C binding**

```
int MPI_Allgather(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

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.

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_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 of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (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) |

**C binding**

```
int MPI_Allgatherv(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, 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. In such a case, sendcount and sendtype are ignored, and 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 of one group (group A) contributes sendcount data items; these data are concatenated and the result is stored at each process in the other group (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 Example using MPI_ALLGATHER

The example in this section uses 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.

```
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) |

**C binding**

```
int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)
```

**F08 binding**

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

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 i-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.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcount and sendtype are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by recvcount and recvtype.

*Rationale.* For large MPI_ALLTOALL instances, allocating both send and receive buffers may consume too much memory. The "in place" option effectively halves the application memory consumption and is useful in situations where the data to be sent will not be used by the sending process after the MPI_ALLTOALL exchange (e.g., in parallel Fast Fourier Transforms). (*End of rationale.*)

*Advice to implementors.* Users may opt to use the "in place" option in order to conserve memory. Quality MPI implementations should thus strive to minimize system buffering. (*End of advice to implementors.*)

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.

*Advice to users.* When a complete exchange 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_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length group size) specifying the number of elements to send to each rank |
| 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 (of length group size) specifying the number of elements that can be received from each rank |
| 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) |

**C binding**

```
int MPI_Alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
             const int recvcounts[], const int rdispls[],
             MPI_Datatype recvtype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
             rdispls, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcounts(*), sdispls(*), recvcounts(*),
    rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, IERROR
```

MPI_ALLTOALLV adds flexibility to MPI_ALLTOALL in that the location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls.

If comm is an intracommunicator, 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 sendcounts[j], sendtype at process i must be equal to the type signature associated with recvcounts[i], 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+sdispls[i]· extent(sendtype),sendcounts[i],sendtype,i,...),

and received a message from every other process with a call to

MPI_Recv(recvbuf+rdispls[i]· extent(recvtype),recvcounts[i],recvtype,i,...).

All arguments on all processes are significant. The argument comm must have identical values on all processes.

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtype are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the recvcounts array and the recvtype, and is taken from the locations of the receive buffer specified by rdispls.

*Advice to users.* Specifying the "in place" option (which must be given on all processes) implies that the same amount and type of data is sent and received between any two processes in the group of the communicator. Different pairs of processes can exchange different amounts of data. Users must ensure that recvcounts[j] and recvtype on process i match recvcounts[i] and recvtype on process j. This symmetric exchange can be useful in applications where the data to be sent will not be used by the sending process after the MPI_ALLTOALLV exchange. (*End of advice to users.*)

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 n independent, point-to-point communications, with two exceptions: all messages use the same datatype, and messages are scattered from (or gathered to) sequential storage. (*End of rationale.*)

*Advice to implementors.* Although the discussion of collective communication in terms of point-to-point operation implies that each message is transferred directly from sender to receiver, implementations may use a tree communication pattern. Messages can be forwarded by intermediate nodes where they are split (for scatter) or concatenated (for gather), if this is more efficient. (*End of advice to implementors.*)

MPI_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length group size) specifying the number of elements to send to each rank |
| 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 | non-negative integer array (of length group size) specifying the number of elements that can be received from each rank |
| 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) |

**C binding**

```
int MPI_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[], MPI_Comm comm)
```

**F08 binding**

```
MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
              rdispls, recvtypes, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcounts(*), sdispls(*), recvcounts(*),
              rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtypes(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
              RDISPLS, RECVTYPES, COMM, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
          RDISPLS(*), RECVTYPES(*), COMM, IERROR
```

MPI_ALLTOALLW is the most general form of complete exchange. Like MPI_TYPE_CREATE_STRUCT, the most general type constructor, MPI_ALLTOALLW allows separate specification of count, displacement and datatype. In addition, to allow maximum flexibility, the displacement of blocks within the send and receive buffers is specified in bytes.

If comm is an intracommunicator, 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 sendcounts[j], sendtypes[j] at process i must be equal to the type signature associated with recvcounts[i], recvtypes[i] at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of 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+sdispls[i],sendcounts[i],sendtypes[i] ,i,...),

and received a message from every other process with a call to

MPI_Recv(recvbuf+rdispls[i],recvcounts[i],recvtypes[i] ,i,...).

All arguments on all processes are significant. The argument comm must describe the same communicator on all processes.

Like for MPI_ALLTOALLV, the "in place" option for intracommunicators is specified by passing MPI_IN_PLACE to the argument sendbuf at *all* processes. In such a case, sendcounts, sdispls and sendtypes are ignored. The data to be sent is taken from the recvbuf and replaced by the received data. Data sent and received must have the same type map as specified by the recvcounts and recvtypes arrays, and is taken from the locations of the receive buffer specified by rdispls.

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 MPI_ALLTOALLW function generalizes several MPI functions by carefully selecting the input arguments. For example, by making all but one process have sendcounts[i] = 0, this achieves an MPI_SCATTERW function. (*End of rationale.*)

## 5.9 Global Reduction Operations

The functions in this section perform a global reduce operation (for example sum, maximum, and logical and) across all members of a group. The reduction operation can be either one of a predefined list of operations, or a user-defined operation. The global reduction functions come in several flavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel prefix) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.

### 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) |

**C binding**

```
int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

**F08 binding**

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

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 of the same length, with elements of the same type as the output buffer at the root. 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 to which each operation can be applied.

In addition, users may define their own operations that can be overloaded to operate on several datatypes, either basic or derived. This is further explained in Section 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. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition.

> *Advice to implementors.* It is strongly recommended that MPI_REDUCE be implemented so that the same result be obtained whenever the function is applied on the same arguments, appearing in the same order. Note that this may prevent optimizations that take advantage of the physical location of ranks. (*End of advice to implementors.*)

> *Advice to users.* Some applications may not be able to ignore the non-associative nature of floating-point operations or may use user-defined operations (see Section 5.9.5) that require a special reduction order and cannot be treated as associative. Such applications should enforce the order of evaluation explicitly. For example, in the case of operations that require a strict left-to-right (or right-to-left) evaluation order, this could be done by gathering all operands at a single process (e.g., with MPI_GATHER), applying the reduction operation in the desired order (e.g., with MPI_REDUCE_LOCAL), and if needed, broadcast or scatter the result to the other processes (e.g., with MPI_BCAST). (*End of advice to users.*)

The datatype argument of MPI_REDUCE must be compatible with op. Predefined operators work only with the MPI types listed in Section 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. It is safest to ensure that the same function is passed to MPI_REDUCE by each process. (*End of advice to users.*)

Overlapping datatypes are permitted in "send" buffers. Overlapping datatypes in "receive" buffers are erroneous and may give unpredictable results.

The "in place" option for intracommunicators is specified by passing the value MPI_IN_PLACE to the argument sendbuf at the root. In such a case, the input data is taken at the root from the receive buffer, where it will be replaced by the output data.

If comm is an 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_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and MPI_REDUCE_LOCAL. These operations are invoked by placing the following in op.

| Name | Meaning |
|---|---|
| MPI_MAX | maximum |
| MPI_MIN | minimum |
| MPI_SUM | sum |
| MPI_PROD | product |
| MPI_LAND | logical and |
| MPI_BAND | bit-wise and |
| MPI_LOR | logical or |
| MPI_BOR | bit-wise or |
| MPI_LXOR | logical exclusive or (xor) |
| MPI_BXOR | bit-wise exclusive or (xor) |
| MPI_MAXLOC | max value and location |
| MPI_MINLOC | min value and location |

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,<br>MPI_UNSIGNED_SHORT, MPI_UNSIGNED,<br>MPI_UNSIGNED_LONG,<br>MPI_LONG_LONG_INT,<br>MPI_LONG_LONG (as synonym),<br>MPI_UNSIGNED_LONG_LONG,<br>MPI_SIGNED_CHAR,<br>MPI_UNSIGNED_CHAR,<br>MPI_INT8_T, MPI_INT16_T,<br>MPI_INT32_T, MPI_INT64_T,<br>MPI_UINT8_T, MPI_UINT16_T,<br>MPI_UINT32_T, and MPI_UINT64_T |
| Fortran integer: | MPI_INTEGER<br>and handles returned from |

| | |
|---|---|
| | MPI_TYPE_CREATE_F90_INTEGER and, if available, MPI_INTEGER1, MPI_INTEGER2, MPI_INTEGER4, MPI_INTEGER8, and MPI_INTEGER16 |
| Floating point: | MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE, and handles returned from MPI_TYPE_CREATE_F90_REAL and, if available, MPI_REAL2, MPI_REAL4, MPI_REAL8, and MPI_REAL16 |
| Logical: | MPI_LOGICAL, MPI_C_BOOL, and MPI_CXX_BOOL |
| Complex: | MPI_COMPLEX, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX (as synonym), MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, MPI_CXX_LONG_DOUBLE_COMPLEX, and handles returned from MPI_TYPE_CREATE_F90_COMPLEX and, if available, MPI_DOUBLE_COMPLEX, MPI_COMPLEX4, MPI_COMPLEX8, MPI_COMPLEX16, and MPI_COMPLEX32 |
| Byte: | MPI_BYTE |
| Multi-language types: | MPI_AINT, MPI_OFFSET, and MPI_COUNT |

Now, the valid datatypes for each operation are specified below.

| Op | Allowed Types |
|---|---|
| MPI_MAX, MPI_MIN | C integer, Fortran integer, Floating point, Multi-language types |
| MPI_SUM, MPI_PROD | C integer, Fortran integer, Floating point, Complex, Multi-language types |
| MPI_LAND, MPI_LOR, MPI_LXOR | C integer, Logical |
| MPI_BAND, MPI_BOR, MPI_BXOR | C integer, Fortran integer, Byte, Multi-language types |

These operations together with all listed datatypes are valid in all supported programming languages, see also Reduce Operations on page 674 in Section 17.2.6.

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
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
END
```

**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
END
```

### 5.9.3 Signed Characters and Reductions

The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR can be used in reduction operations. MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER (which represent printable characters) cannot be used in reduction operations. In a heterogeneous environment, MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER will be translated so as to preserve the printable character, whereas MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR will be translated so as to preserve the integer value.

*Advice to users.* The types MPI_CHAR, MPI_WCHAR, and MPI_CHARACTER are intended for characters, and so will be translated to preserve the printable representation, rather than the integer value, if sent between machines with different character codes. The types MPI_SIGNED_CHAR and MPI_UNSIGNED_CHAR should be used in C if the integer value should be preserved. (*End of advice to users.*)

### 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 $u = \max_i u_i$ 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 with op = MPI_MAXLOC will return the maximum value and the rank of the first process with that value. Similarly, MPI_MINLOC can be used to return a minimum and its index. More generally, MPI_MINLOC computes a *lexicographic minimum*, where elements are ordered according to the first component of each pair, and ties are resolved according to the second component.

The 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 predefined datatypes. The operations MPI_MAXLOC and MPI_MINLOC can be used with each of the following datatypes.

Fortran:

| Name | Description |
|---|---|
| MPI_2REAL | pair of `REAL`s |
| MPI_2DOUBLE_PRECISION | pair of `DOUBLE PRECISION` variables |
| MPI_2INTEGER | pair of `INTEGER`s |

C:

| Name | Description |
|---|---|
| MPI_FLOAT_INT | `float` and `int` |
| MPI_DOUBLE_INT | `double` and `int` |
| MPI_LONG_INT | `long` and `int` |
| MPI_2INT | pair of `int` |
| MPI_SHORT_INT | `short` and `int` |
| MPI_LONG_DOUBLE_INT | `long double` and `int` |

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_SHORT_INT is *as if* defined by the following sequence of instructions.

```
struct mystruct {
    short val;
    int rank;
};
type[0] = MPI_SHORT;
type[1] = MPI_INT;
disp[0] = 0;
disp[1] = offsetof(struct mystruct, rank);
block[0] = 1;
block[1] = 1;
MPI_Type_create_struct(2, block, disp, type, MPI_SHORT_INT);
```

Similar statements apply for MPI_FLOAT_INT, MPI_LONG_INT and MPI_DOUBLE_INT.

The following examples use intracommunicators.

**Example 5.17**

Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

```
...
/* each process has an array of 30 double: ain[30]
 */
double ain[30], aout[30];
int  ind[30];
struct {
    double val;
    int   rank;
} in[30], out[30];
int i, myrank, root;

MPI_Comm_rank(comm, &myrank);
for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}
MPI_Reduce(in, out, 30, MPI_DOUBLE_INT, MPI_MAXLOC, root, comm);
/* At this point, the answer resides on process root
 */
if (myrank == root) {
    /* read ranks out
     */
    for (i=0; i<30; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
    }
}
```

**Example 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_2DOUBLE_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

MPI_OP_CREATE(user_fn, commute, op)

| | | |
|---|---|---|
| IN | user_fn | user defined function (function) |
| IN | commute | true if commutative; false otherwise. |
| OUT | op | operation (handle) |

**C binding**

```
int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)
```

**F08 binding**

```
MPI_Op_create(user_fn, commute, op, ierror)
    PROCEDURE(MPI_User_function) :: user_fn
    LOGICAL, INTENT(IN) :: commute
    TYPE(MPI_Op), INTENT(OUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
    EXTERNAL USER_FN
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

MPI_OP_CREATE binds a user-defined reduction operation to an op handle that can subsequently be used in MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_EXSCAN, all nonblocking variants of those (see Section 5.12), and MPI_REDUCE_LOCAL. The user-defined operation is assumed to be associative. If commute = true, then the operation should be both commutative and associative. If commute = false, then the order of operands is fixed and is defined to be in ascending, process rank order, beginning with process zero. The order of evaluation can be changed, talking advantage of the associativity of the operation. If commute = true then the order of evaluation can be changed, taking advantage of commutativity and associativity.

The argument user_fn is the user-defined function, which must have the following four arguments: invec, inoutvec, len, and datatype.

The ISO C prototype for the function is the following.

```
typedef void MPI_User_function(void* invec, void* inoutvec, int *len,
             MPI_Datatype *datatype);
```

The Fortran declarations of the user-defined function user_fn appear below.

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
      USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE ::  invec, inoutvec
      INTEGER ::  len
      TYPE(MPI_Datatype) ::  datatype

SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
    <type> INVEC(LEN), INOUTVEC(LEN)
    INTEGER LEN, 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]∘v[i], for i=0 , ... , len-1, where ∘ is the reduce operation that the function computes.

Informally, we can think of invec and inoutvec as arrays of len elements that user_fn is combining. The result of the reduction over-writes values in inoutvec, hence the name. Each invocation of the function results in the pointwise evaluation of the reduce operator on len elements: i.e., the function returns in inoutvec[i] the value invec[i] ∘ inoutvec[i], for i=0, ... , count-1, where ∘ is the combining operation computed by the function.

*Rationale.* The len argument allows MPI_REDUCE to avoid calling the function for each element in the input buffer. Rather, the system can choose to apply the function to chunks of input. In C, it is passed in as a reference for reasons of compatibility with Fortran.

By internally comparing the value of the datatype argument to known, global handles, it is possible to overload the use of a single user-defined function for several, different 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. 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 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 datatype argument that it is passed, and cannot identify, by itself, the correspondence between the datatype handles and the datatype they represent. This correspondence was established when the datatypes were created. Before the library is used, a library initialization preamble must be executed. This preamble code will define the datatypes that are used by the library, and store handles to these datatypes in global, static variables that are shared by the user code and the library code.

The Fortran version of MPI_REDUCE will invoke a user-defined reduce function using the Fortran calling conventions and will pass a Fortran-type datatype argument; the C version will use C calling convention and the C representation of a 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 MPI_REDUCE not supporting the "in place" option.

```
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
 */
if (rank == root) {
    MPI_Irecv(recvbuf, count, datatype, groupsize-1,..., &req);
}
if (rank == groupsize-1) {
    MPI_Send(sendbuf, count, datatype, root, ...);
}
if (rank == root) {
    MPI_Wait(&req, &status);
}
```

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_reduce(). 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 len <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.*)

MPI_OP_FREE(op)

| | | |
|---|---|---|
| INOUT | op | operation (handle) |

**C binding**

```
int MPI_Op_free(MPI_Op *op)
```

**F08 binding**

```
MPI_Op_free(op, ierror)
    TYPE(MPI_Op), INTENT(INOUT) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_OP_FREE(OP, IERROR)
    INTEGER OP, IERROR
```

Marks a user-defined reduction operation for deallocation and sets op to MPI_OP_NULL.

### 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(void *inP, void *inoutP, int *len, MPI_Datatype *dptr)
{
    int i;
    Complex c;
    Complex *in = (Complex *)inP, *inout = (Complex *)inoutP;

    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;
        in++; inout++;
    }
```

```
}

/* and, to call it...
 */
...

    /* each process has an array of 100 Complexes
     */
    Complex a[100], answer[100];
    MPI_Op myOp;
    MPI_Datatype ctype;

    /* explain to MPI how type Complex is defined
     */
    MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
    MPI_Type_commit(&ctype);
    /* create the complex-product user-op
     */
    MPI_Op_create(myProd, 1, &myOp);

    MPI_Reduce(a, answer, 100, ctype, myOp, root, comm);

    /* At this point, the answer, which consists of 100 Complexes,
     * resides on process root
     */
```

**Example 5.21** How to use the mpi_f08 interface of the Fortran MPI_User_function.

```
  subroutine my_user_function(invec, inoutvec, len, type)   bind(c)
    use, intrinsic :: iso_c_binding, only : c_ptr, c_f_pointer
    use mpi_f08
    type(c_ptr), value :: invec, inoutvec
    integer :: len
    type(MPI_Datatype) :: type
    real, pointer :: invec_r(:), inoutvec_r(:)
    if (type%MPI_VAL == MPI_REAL%MPI_VAL) then
       call c_f_pointer(invec, invec_r, (/ len /))
       call c_f_pointer(inoutvec, inoutvec_r, (/ len /))
       inoutvec_r = invec_r + inoutvec_r
    end if
  end subroutine
```

### 5.9.6 All-Reduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.

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) |

**C binding**

```
int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

**F08 binding**

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

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 5.22**

A routine that computes the product of a vector and an array that are distributed across a group of processes and returns the answer at all nodes (see also Example 5.16).

```
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_ALLREDUCE(sum, c, n, MPI_REAL, MPI_SUM, comm, ierr)

! return result at all nodes
RETURN
END
```

### 5.9.7 Process-Local Reduction

The functions in this section are of importance to library implementors who may want to implement special reduction patterns that are otherwise not easily covered by the standard MPI operations.

The following function applies a reduction operator to local arguments.

MPI_REDUCE_LOCAL(inbuf, inoutbuf, count, datatype, op)

| | | |
|---|---|---|
| IN | inbuf | input buffer (choice) |
| INOUT | inoutbuf | combined input and output buffer (choice) |
| IN | count | number of elements in inbuf and inoutbuf buffers (non-negative integer) |
| IN | datatype | data type of elements of inbuf and inoutbuf buffers (handle) |
| IN | op | operation (handle) |

**C binding**

```
int MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,
             MPI_Datatype datatype, MPI_Op op)
```

**F08 binding**

```
MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    TYPE(*), DIMENSION(..)  ::  inoutbuf
    INTEGER, INTENT(IN) ::  count
```

```
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
    <type> INBUF(*), INOUTBUF(*)
    INTEGER COUNT, DATATYPE, OP, IERROR
```

The function applies the operation given by op element-wise to the elements of inbuf and inoutbuf with the result stored element-wise in inoutbuf, as explained for user-defined operations in Section 5.9.5. Both inbuf and inoutbuf (input as well as result) have the same number of elements given by count and the same datatype given by datatype. The MPI_IN_PLACE option is not allowed.

Reduction operations can be queried for their commutativity.

MPI_OP_COMMUTATIVE(op, commute)

| | | |
|---|---|---|
| IN | op | operation (handle) |
| OUT | commute | true if op is commutative, false otherwise (logical) |

**C binding**

```
int MPI_Op_commutative(MPI_Op op, int *commute)
```

**F08 binding**

```
MPI_Op_commutative(op, commute, ierror)
    TYPE(MPI_Op), INTENT(IN) :: op
    LOGICAL, INTENT(OUT) :: commute
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
    LOGICAL COMMUTE
    INTEGER OP, IERROR
```

## 5.10 Reduce-Scatter

MPI includes variants of the reduce operations where the result is scattered to all processes in a group on return. One variant scatters equal-sized blocks to all processes, while another variant scatters blocks that may vary in size for each process.

### 5.10.1 MPI_REDUCE_SCATTER_BLOCK

MPI_REDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | element count per block (non-negative integer) |
| IN | datatype | data type of elements of send and receive buffers (handle) |
| IN | op | operation (handle) |
| IN | comm | communicator (handle) |

**C binding**

```
int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,
             int recvcount, MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm)
```

**F08 binding**

```
MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
```

If comm is an intracommunicator, MPI_REDUCE_SCATTER_BLOCK first performs a global, element-wise reduction on vectors of count = n*recvcount elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcount, datatype, op and comm. The resulting vector is treated as n consecutive blocks of recvcount elements that are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcount, and datatype.

*Advice to implementors.* The MPI_REDUCE_SCATTER_BLOCK routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to recvcount*n, followed by an MPI_SCATTER with sendcount equal to recvcount. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument on *all* processes. In this case, the input data is taken from the receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B) and vice versa. Within each group, all processes provide the same value for the recvcount argument, and provide input vectors of count = n*recvcount elements stored in the send buffers, where n is the size of the group. The number of elements count must be the same for the two groups. The resulting vector from the other group is scattered in blocks of recvcount elements among the processes in the group.

*Rationale.* The last restriction is needed so that the length of the send buffer of one group can be determined by the local recvcount argument of the other group. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

### 5.10.2 MPI_REDUCE_SCATTER

MPI_REDUCE_SCATTER extends the functionality of MPI_REDUCE_SCATTER_BLOCK such that the scattered blocks can vary in size. Block sizes are determined by the recvcounts array, such that the i-th block contains recvcounts[i] elements.

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 (of length group size) specifying the number of elements of the result distributed to each process. |
| IN | datatype | data type of elements of send and receive buffers (handle) |
| IN | op | operation (handle) |
| IN | comm | communicator (handle) |

**C binding**

```
int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,
             const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm)
```

**F08 binding**

```
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

If comm is an intracommunicator, MPI_REDUCE_SCATTER first performs a global, element-wise reduction on vectors of count $= \sum_{i=0}^{n-1}$ recvcounts[i] elements in the send buffers defined by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buffer defined by recvbuf, recvcounts[i] and datatype.

> *Advice to implementors.* The MPI_REDUCE_SCATTER routine is functionally equivalent to: an MPI_REDUCE collective operation with count equal to the sum of recvcounts[i] followed by MPI_SCATTERV with sendcounts equal to recvcounts. However, a direct implementation may run faster. (*End of advice to implementors.*)

The "in place" option for intracommunicators is specified by passing MPI_IN_PLACE in the sendbuf argument. In this case, the input data is taken from the receive buffer. It is not required to specify the "in place" option on all processes, since the processes for which recvcounts[i] ==0 may not have allocated a receive buffer.

If comm is an intercommunicator, then the result of the reduction of the data provided by processes in one group (group A) is scattered among processes in the other group (group B), and vice versa. Within each group, all processes provide the same recvcounts argument, and provide input vectors of count $= \sum_{i=0}^{n-1}$ recvcounts[i] elements stored in the send buffers, where n is the size of the group. The resulting vector from the other group is scattered in blocks of recvcounts[i] elements among the processes in the group. The number of elements count must be the same for the two groups.

> *Rationale.* The last restriction is needed so that the length of the send buffer can be determined by the sum of the local recvcounts entries. Otherwise, a communication is needed to figure out how many elements are reduced. (*End of rationale.*)

## 5.11 Scan

### 5.11.1 Inclusive Scan

MPI_SCAN(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 | communicator (handle) |

**C binding**

```
int MPI_Scan(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

**F08 binding**

```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

If comm is an intracommunicator, 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 routine is called by all group members using the same arguments for count, datatype, op and comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE. The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE.

The "in place" option for intracommunicators 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.

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) |

**C binding**

```
int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

**F08 binding**

```
MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

If comm is an intracommunicator, 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 signficant 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 $0, \ldots, i-1$ (inclusive). The routine is called by all group members using the same arguments for count, datatype, op and comm, except that for user-defined operations, the same rules apply as for MPI_REDUCE. The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for MPI_REDUCE.

The "in place" option for intracommunicators 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. The receive buffer on rank 0 is not changed by this operation.

This operation is invalid for intercommunicators.

*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. (*End of rationale.*)

### 5.11.3 Example using MPI_SCAN

The example in this section uses an intracommunicator.

**Example 5.23**

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* | $v_1$ | $v_1+v_2$ | $v_3$ | $v_3+v_4$ | $v_3+v_4+v_5$ | $v_6$ | $v_6+v_7$ | $v_8$ |

The operator that produces this effect is

$$\begin{pmatrix} u \\ i \end{pmatrix} \circ \begin{pmatrix} v \\ j \end{pmatrix} = \begin{pmatrix} w \\ j \end{pmatrix},$$

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.

```
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;
        in++; inout++;
    }
}
```

Note that the inout argument to the user-defined function corresponds to the right-hand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```
int i,base;
SegScanPair  a, answer;
MPI_Op       myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
MPI_Aint     disp[2];
int          blocklen[2] = { 1, 1};
MPI_Datatype sspair;

/* explain to MPI how type SegScanPair is defined
 */
MPI_Get_address(&a, disp);
MPI_Get_address(&a.log, disp+1);
base = disp[0];
for (i=0; i<2; ++i) disp[i] -= base;
MPI_Type_create_struct(2, blocklen, disp, type, &sspair);
MPI_Type_commit(&sspair);
/* create the segmented-scan user-op
 */
MPI_Op_create(segScan, 0, &myOp);
...
MPI_Scan(&a, &answer, 1, sspair, myOp, comm);
```

## 5.12 Nonblocking Collective Operations

As described in Section 3.7, performance of many applications can be improved by overlapping communication and computation, and many systems enable this. Nonblocking collective operations combine the potential benefits of nonblocking point-to-point operations, to exploit overlap and to avoid synchronization, with the optimized implementation and message scheduling provided by collective operations [30, 34]. One way of doing this would be to perform a blocking collective operation in a separate thread. An alternative mechanism that often leads to better performance (e.g., avoids context switching, scheduler overheads, and thread management) is to use nonblocking collective communication [32].

The nonblocking collective communication model is similar to the model used for nonblocking point-to-point communication. A nonblocking call initiates a collective operation, which must be completed in a separate completion call. Once initiated, the operation may progress independently of any computation or other communication at participating processes. In this manner, nonblocking collective operations can mitigate possible synchronizing effects of collective operations by running them in the "background." In addition to enabling communication-computation overlap, nonblocking collective operations can perform collective operations on overlapping communicators, which would lead to deadlocks with blocking operations. Their semantic advantages can also be useful in combination with point-to-point communication.

As in the nonblocking point-to-point case, all calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation, which indicates that the system may start to copy data out of the send buffer and into the receive buffer. Once initiated, all associated send buffers and buffers associated with input arguments (such as arrays of counts, displacements, or datatypes in the vector versions of the collectives) should not be modified, and all associated receive buffers should not be accessed, until the collective operation completes. The call returns a request handle, which must be passed to a completion call.

All completion calls (e.g., MPI_WAIT) described in Section 3.7.3 are supported for nonblocking collective operations. Similarly to the blocking case, nonblocking collective operations are considered to be complete when the local part of the operation is finished, i.e., for the caller, the semantics of the operation are guaranteed and all buffers can be safely accessed and modified. Completion does not indicate that other processes have completed or even started the operation (unless otherwise implied by the description of the operation). Completion of a particular nonblocking collective operation also does not indicate completion of any other posted nonblocking collective (or send-receive) operations, whether they are posted before or after the completed operation.

*Advice to users.* Users should be aware that implementations are allowed, but not required (with exception of MPI_IBARRIER), to synchronize processes during the completion of a nonblocking collective operation. (*End of advice to users.*)

Upon returning from a completion call in which a nonblocking collective operation completes, the MPI_ERROR field in the associated status object is set appropriately, see Section 3.2.5 on page 32. The values of the MPI_SOURCE and MPI_TAG fields are undefined. It is valid to mix different request types (i.e., any combination of collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with a nonblocking collective operation. Nonblocking collective requests created using the APIs described in this section are not persistent. However, persistent collective requests can be created using persistent collective operations described in Sections 5.13 and 7.8.

*Rationale.* Freeing an active nonblocking collective request could cause similar problems as discussed for point-to-point requests (see Section 3.7.3). Cancelling a request is not supported because the semantics of this operation are not well-defined. (*End of rationale.*)

Multiple nonblocking collective operations can be outstanding on a single communicator. If the nonblocking call causes some system resource to be exhausted, then it will fail and generate an MPI exception. Quality implementations of MPI should ensure that this happens only in pathological cases. That is, an MPI implementation should be able to support a large number of pending nonblocking operations.

Unlike point-to-point operations, nonblocking collective operations do not match with blocking collective operations, and collective operations do not have a tag argument. All processes must call collective operations (blocking and nonblocking) in the same order per communicator. In particular, once a process calls a collective operation, all other processes in the communicator must eventually call the same collective operation, and no other collective operation with the same communicator in between. This is consistent with the ordering rules for blocking collective operations in threaded environments.

*Rationale.* Matching blocking and nonblocking collective operations is not allowed because the implementation might use different communication algorithms for the two cases. Blocking collective operations may be optimized for minimal time to completion, while nonblocking collective operations may balance time to completion with CPU overhead and asynchronous progression.

The use of tags for collective operations can prevent certain hardware optimizations. (*End of rationale.*)

*Advice to users.* If program semantics require matching blocking and nonblocking collective operations, then a nonblocking collective operation can be initiated and immediately completed with a blocking wait to emulate blocking behavior. (*End of advice to users.*)

In terms of data movement, each nonblocking collective operation has the same effect as its blocking counterpart for intracommunicators and intercommunicators after completion. Likewise, upon completion, nonblocking collective reduction operations have the same effect as their blocking counterparts, and the same restrictions and recommendations on reduction orders apply.

The use of the "in place" option is allowed exactly as described for the corresponding blocking collective operations. When using the "in place" option, message buffers function as both send and receive buffers. Such buffers should not be modified or accessed until the operation completes.

Progression rules for nonblocking collective operations are similar to progression of nonblocking point-to-point operations, refer to Section 3.7.4.

*Advice to implementors.* Nonblocking collective operations can be implemented with local execution schedules [33] using nonblocking point-to-point communication and a reserved tag-space. (*End of advice to implementors.*)

### 5.12.1 Nonblocking Barrier Synchronization

MPI_IBARRIER(comm ,request)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ibarrier(comm, request, ierror)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IBARRIER(COMM, REQUEST, IERROR)
    INTEGER COMM, REQUEST, IERROR
```

MPI_IBARRIER is a nonblocking version of MPI_BARRIER. By calling MPI_IBARRIER, a process notifies that it has reached the barrier. The call returns immediately, independent of whether other processes have called MPI_IBARRIER. The usual barrier semantics are enforced at the corresponding completion operation (test or wait), which in the intra-communicator case will complete only after all other processes in the communicator have called MPI_IBARRIER. In the intercommunicator case, it will complete when all processes in the remote group have called MPI_IBARRIER.

> *Advice to users.* A nonblocking barrier can be used to hide latency. Moving independent computations between the MPI_IBARRIER and the subsequent completion call can overlap the barrier latency and therefore shorten possible waiting times. The semantic properties are also useful when mixing collective operations and point-to-point messages. (*End of advice to users.*)

### 5.12.2 Nonblocking Broadcast

MPI_IBCAST(buffer, count, datatype, root, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_BCAST (see Section 5.4).

#### Example using MPI_IBCAST

The example in this section uses an intracommunicator.

**Example 5.24**

Start a broadcast of 100 `int`s from process `0` to every process in the group, perform some computation on independent data, and then complete the outstanding broadcast operation.

```
MPI_Comm comm;
int array1[100], array2[100];
int root=0;
MPI_Request req;
...
MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req);
compute(array2, 100);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

### 5.12.3 Nonblocking Gather

MPI_IGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request)

| | | |
|---|---|---|
| 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, significant only at root) |
| IN | recvcount | number of elements for any single receive (non-negative integer, significant only at root) |
| IN | recvtype | data type of recv buffer elements (significant only at root) (handle) |
| IN | root | rank of receiving process (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
             MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
              IERROR
```

This call starts a nonblocking variant of MPI_GATHER (see Section 5.5).

MPI_IGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, request)

| | | |
|---|---|---|
| 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, significant only at root) |
| IN | recvcounts | non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root) |
| IN | displs | integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root) |
| IN | recvtype | data type of recv buffer elements (significant only at root) (handle) |
| IN | root | rank of receiving process (integer) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, const int recvcounts[], const int displs[],
```

```
            MPI_Datatype recvtype, int root, MPI_Comm comm,
            MPI_Request *request)
```

**F08 binding**

```
MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
            recvtype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
            RECVTYPE, ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
             COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_GATHERV (see Section 5.5).

### 5.12.4 Nonblocking Scatter

MPI_ISCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | address of send buffer (choice, significant only at root) |
| IN | sendcount | number of elements sent to each process (non-negative integer, significant only at root) |
| IN | sendtype | data type of send buffer elements (significant only at root) (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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
            MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
              IERROR
```

This call starts a nonblocking variant of MPI_SCATTER (see Section 5.6).

MPI_ISCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, request)

| | | |
|---|---|---|
| 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 rank |
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Iscatterv(const void* sendbuf, const int sendcounts[],
             const int displs[], MPI_Datatype sendtype, void* recvbuf,
             int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
             recvtype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), displs(*)
    INTEGER, INTENT(IN) ::  recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
              COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_SCATTERV (see Section 5.6).

### 5.12.5 Nonblocking Gather-to-all

MPI_IALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Iallgather(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
```

```
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_ALLGATHER (see Section 5.7).

MPI_IALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Iallgatherv(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Request* request)
```

**F08 binding**

```
MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_ALLGATHERV (see Section 5.7).

### 5.12.6 Nonblocking All-to-All Scatter/Gather

MPI_IALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ialltoall(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
              COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_ALLTOALL (see Section 5.8).

MPI_IALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length group size) specifying the number of elements to send to each rank |
| 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 (of length group size) specifying the number of elements that can be received from each rank |
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[],
              const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
              const int recvcounts[], const int rdispls[],
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
              rdispls, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
               recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
              RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_ALLTOALLV (see Section 5.8).

MPI_IALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers) |
| IN | sdispls | integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers) |
| IN | sendtypes | array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles) |
| OUT | recvbuf | address of receive buffer (choice) |
| IN | recvcounts | integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers) |
| IN | rdispls | integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers) |
| IN | recvtypes | array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ialltoallw(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[], MPI_Comm comm,
              MPI_Request *request)
```

**F08 binding**

```
MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS ::  sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
              RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_ALLTOALLW (see Section 5.8).

### 5.12.7 Nonblocking Reduce

MPI_IREDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
```

```
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_REDUCE (see Section 5.9.1).

*Advice to implementors.* The implementation is explicitly allowed to use different algorithms for blocking and nonblocking reduction operations that might change the order of evaluation of the operations. However, as for MPI_REDUCE, it is strongly recommended that MPI_IREDUCE 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 processes. (*End of advice to implementors.*)

*Advice to users.* For operations which are not truly associative, the result delivered upon completion of the nonblocking reduction may not exactly equal the result delivered by the blocking reduction, even when specifying the same arguments in the same order. (*End of advice to users.*)

### 5.12.8 Nonblocking All-Reduce

MPI_IALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_ALLREDUCE (see Section 5.9.6).

### 5.12.9 Nonblocking Reduce-Scatter with Equal Blocks

MPI_IREDUCE_SCATTER_BLOCK(sendbuf, recvbuf, recvcount, datatype, op, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | element count per block (non-negative integer) |
| IN | datatype | data type of elements of send and receive buffers (handle) |
| IN | op | operation (handle) |
| IN | comm | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf,
             int recvcount, MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
```

```
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_REDUCE_SCATTER_BLOCK (see Section 5.10.1).

### 5.12.10 Nonblocking Reduce-Scatter

MPI_IREDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf,
             const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             REQUEST, IERROR)
```

```
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_REDUCE_SCATTER (see Section 5.10.2).

### 5.12.11 Nonblocking Inclusive Scan

MPI_ISCAN(sendbuf, recvbuf, count, datatype, op, comm, request)

| | | |
|---|---|---|
| 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 | communicator (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Iscan(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_SCAN (see Section 5.11).

### 5.12.12 Nonblocking Exclusive Scan

MPI_IEXSCAN(sendbuf, recvbuf, count, datatype, op, comm, request)

| | | |
|---|---|---|
| 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) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_EXSCAN (see Section 5.11.2).

## 5.13 Persistent Collective Operations

Many parallel computation algorithms involve repetitively executing a collective communication operation with the same arguments each time. As with persistent point-to-point operations (see Section 3.9), persistent collective operations allow the MPI programmer to specify operations that will be reused frequently (with fixed arguments). MPI can be designed to select a more efficient way to perform the collective operation based on the parameters specified when the operation is initialized. This "planned-transfer" approach can offer significant performance benefits for programs with repetitive communication patterns.

In terms of data movement, each persistent collective operation has the same effect as its blocking and nonblocking counterparts for intracommunicators and intercommunicators after completion. Likewise, upon completion, persistent collective reduction operations perform the same operation as their blocking and nonblocking counterparts, and the same restrictions and recommendations on reduction orders apply (see also Section 5.9.1).

Initialization calls for MPI persistent collective operations are non-local and follow all the existing rules for collective operations, in particular ordering; programs that do not conform to these restrictions are erroneous. After initialization, all arrays associated with input arguments (such as arrays of counts, displacements, and datatypes in the vector versions of the collectives) must not be modified until the corresponding persistent request is freed with MPI_REQUEST_FREE.

The request argument is an output argument that can be used zero or more times with MPI_START or MPI_STARTALL in order to start the collective operation. The request is initially inactive after the initialization call. Once initialized, persistent collective operations can be started in any order and the order can differ among processes in the communicator.

*Rationale.* All ordering requirements that an implementation may need to match up collective operations across the communicator are achieved through the ordering requirements of the initialization functions. This enables out-of-order starts for the persistent operations, and particularly supports their use in MPI_STARTALL. (*End of rationale.*)

*Advice to implementors.* An MPI implementation should do no worse than duplicating the communicator during the initialization function, caching the input arguments, and calling the appropriate nonblocking collective function, using the cached arguments, during MPI_START. High-quality implementations should be able to amortize setup costs and further optimize by taking advantage of early-binding, such as efficient and effective pre-allocation of certain resources and algorithm selection. (*End of advice to implementors.*)

A request must be inactive when it is started. Starting the operation makes the request active. Once any process starts a persistent collective operation, it must complete that operation and all other processes in the communicator must eventually start (and complete) the same persistent collective operation. Persistent collective operations cannot be matched with blocking or nonblocking collective operations. Completion of a persistent collective operation makes the corresponding request inactive. After starting a persistent collective operation, all associated send buffers must not be modified and all associated receive buffers must not be accessed until the corresponding persistent request is completed.

Completing a persistent collective request, for example using MPI_TEST or MPI_WAIT, makes it inactive, but does not free the request. This is the same behavior as for persistent point-to-point requests. Inactive persistent collective requests can be freed using MPI_REQUEST_FREE. It is erroneous to free an active persistent collective request. Persistent collective operations cannot be canceled; it is erroneous to use MPI_CANCEL on a persistent collective request.

For every nonblocking collective communication operation in MPI, there is a corresponding persistent collective operation with the analogous API signature.

The collective persistent API signatures include an MPI_INFO object in order to support optimization hints and other information that may be non-standard. Persistent collective operations may be optimized during communicator creation or by the initialization operation of an individual persistent collective. Note that communicator-scoped hints should be provided using MPI_COMM_SET_INFO while, for operation-scoped hints, they are supplied to the persistent collective communication initialization functions using the info argument.

### 5.13.1 Persistent Barrier Synchronization

MPI_BARRIER_INIT(comm, info, request)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Barrier_init(comm, info, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
    INTEGER COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the barrier operation.

### 5.13.2 Persistent Broadcast

MPI_BCAST_INIT(buffer, count, datatype, root, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Bcast_init(void* buffer, int count, MPI_Datatype datatype,
             int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the broadcast operation.

### 5.13.3 Persistent Gather

MPI_GATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request)

| | | |
|---|---|---|
| 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, significant only at root) |
| IN | recvcount | number of elements for any single receive (non-negative integer, significant only at root) |
| IN | recvtype | data type of recv buffer elements (significant only at root) (handle) |
| IN | root | rank of receiving process (integer) |
| IN | comm | communicator (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Gather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
              REQUEST, IERROR
```

Creates a persistent collective communication request for the gather operation.

MPI_GATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, info, request)

| | | |
|---|---|---|
| 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, significant only at root) |
| IN | recvcounts | non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root) |
| IN | displs | integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root) |
| IN | recvtype | data type of recv buffer elements (significant only at root) (handle) |
| IN | root | rank of receiving process (integer) |
| IN | comm | communicator (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Gatherv_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, int root,
             MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
              COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the gatherv operation.

### 5.13.4 Persistent Scatter

MPI_SCATTER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | address of send buffer (choice, significant only at root) |
| IN | sendcount | number of elements sent to each process (non-negative integer, significant only at root) |
| IN | sendtype | data type of send buffer elements (significant only at root) (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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Scatter_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
              REQUEST, IERROR
```

Creates a persistent collective communication request for the scatter operation.

MPI_SCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request)

| | | |
|---|---|---|
| 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 rank |
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Scatterv_init(const void* sendbuf, const int sendcounts[],
             const int displs[], MPI_Datatype sendtype, void* recvbuf,
             int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
             recvcount, recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), displs(*)
    INTEGER, INTENT(IN) ::  recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
              COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the scatterv operation.

### 5.13.5 Persistent Gather-to-all

MPI_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Allgather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR
```

Creates a persistent collective communication request for the allgather operation.

MPI_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Allgatherv_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Info info, MPI_Request* request)
```

**F08 binding**

```
MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
             displs, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the allgatherv operation.

### 5.13.6 Persistent All-to-All Scatter/Gather

MPI_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Alltoall_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR
```

Creates a persistent collective communication request for the alltoall operation.

MPI_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length group size) specifying the number of elements to send to each rank |
| 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 (of length group size) specifying the number of elements that can be received from each rank |
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Alltoallv_init(const void* sendbuf, const int sendcounts[],
              const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
              const int recvcounts[], const int rdispls[],
              MPI_Datatype recvtype, MPI_Comm comm, MPI_info info,
              MPI_Request *request)
```

**F08 binding**

```
MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
              recvcounts, rdispls, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), sdispls(*),
               recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
              RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the alltoallv operation.

MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers) |
| IN | sdispls | integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers) |
| IN | sendtypes | array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles) |
| OUT | recvbuf | address of receive buffer (choice) |
| IN | recvcounts | integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers) |
| IN | rdispls | integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers) |
| IN | recvtypes | array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles) |
| IN | comm | communicator (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Alltoallw_init(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[], MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
              RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the alltoallw operation.

### 5.13.7 Persistent Reduce

MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Reduce_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count, root
```

```
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the reduce operation.

### 5.13.8 Persistent All-Reduce

MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the allreduce operation.

### 5.13.9 Persistent Reduce-Scatter with Equal Blocks

MPI_REDUCE_SCATTER_BLOCK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | element count per block (non-negative integer) |
| IN | datatype | data type of elements of send and receive buffers (handle) |
| IN | op | operation (handle) |
| IN | comm | communicator (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Reduce_scatter_block_init(const void* sendbuf, void* recvbuf,
             int recvcount, MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
             comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
             COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the reduce-scatter with equal blocks operation.

### 5.13.10 Persistent Reduce-Scatter

MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Reduce_scatter_init(const void* sendbuf, void* recvbuf,
             const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the reduce-scatter operation.

### 5.13.11 Persistent Inclusive Scan

MPI_SCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)

| | | |
|---|---|---|
| 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 | communicator (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Scan_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the inclusive scan operation.

### 5.13.12 Persistent Exclusive Scan

MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)

| | | |
|---|---|---|
| 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) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Exscan_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the exclusive scan operation.

## 5.14 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 5.25**

The following is erroneous.

```
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 5.26**

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 dependencies occur. Nonblocking collective operations can alleviate this issue.

**Example 5.27**

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;
}
```

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 5.28**

An unsafe, non-deterministic program.

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

All three processes participate in a broadcast. Process 0 sends a message to process 1 after the broadcast, and process 2 sends a message to process 1 before the broadcast. Process 1 receives before and after the broadcast, with a wildcard source argument.

Two possible executions of this program, with different matchings of sends and receives, are illustrated in Figure 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.

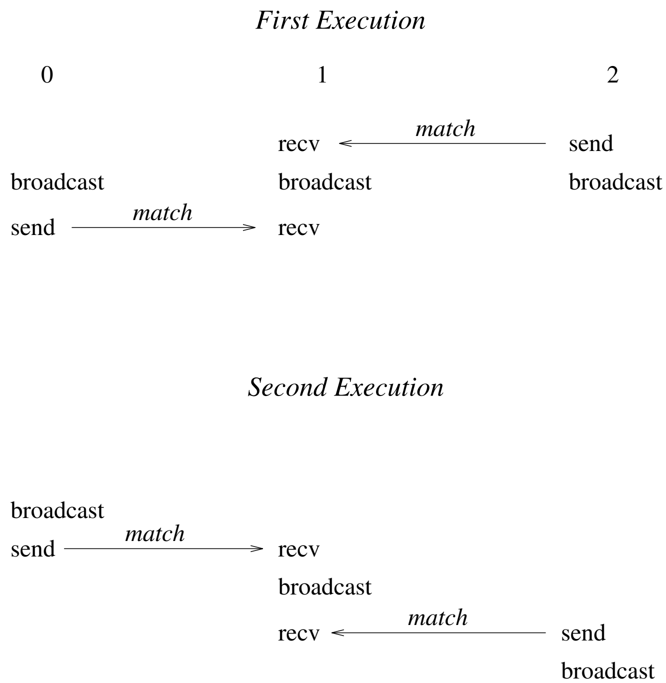


Figure 5.12: A race condition causes non-deterministic matching of sends and receives. One cannot rely on synchronization from a broadcast to make the program deterministic.

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 responsibility 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.*)

**Example 5.29**

Blocking and nonblocking collective operations can be interleaved, i.e., a blocking collective operation can be posted even if there is a nonblocking collective operation outstanding.

```
MPI_Request req;

MPI_Ibarrier(comm, &req);
MPI_Bcast(buf1, count, type, 0, comm);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Each process starts a nonblocking barrier operation, participates in a blocking broadcast and then waits until every other process started the barrier operation. This effectively turns the broadcast into a synchronizing broadcast with possible communication/communication overlap (MPI_Bcast is allowed, but not required to synchronize).

**Example 5.30**

The starting order of collective operations on a particular communicator defines their matching. The following example shows an erroneous matching of different collective operations on the same communicator.

```
MPI_Request req;
switch(rank) {
    case 0:
        /* erroneous matching */
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        /* erroneous matching */
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Ibarrier(comm, &req);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

This ordering would match MPI_Ibarrier on rank 0 with MPI_Bcast on rank 1 which is erroneous and the program behavior is undefined. However, if such an order is required, the user must create different duplicate communicators and perform the operations on them. If started with two processes, the following program would be correct:

```
MPI_Request req;
MPI_Comm dupcomm;
MPI_Comm_dup(comm, &dupcomm);
switch(rank) {
    case 0:
        MPI_Ibarrier(comm, &req);
        MPI_Bcast(buf1, count, type, 0, dupcomm);
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
    case 1:
        MPI_Bcast(buf1, count, type, 0, dupcomm);
        MPI_Ibarrier(comm, &req);
```

```
        MPI_Wait(&req, MPI_STATUS_IGNORE);
        break;
}
```

*Advice to users.* The use of different communicators offers some flexibility regarding the matching of nonblocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (*End of advice to users.*)

**Example 5.31**

Nonblocking collective operations can rely on the same progression rules as nonblocking point-to-point messages. Thus, if started with two processes, the following program is a valid MPI program and is guaranteed to terminate:

```
MPI_Request req;

switch(rank) {
    case 0:
      MPI_Ibarrier(comm, &req);
      MPI_Wait(&req, MPI_STATUS_IGNORE);
      MPI_Send(buf, count, dtype, 1, tag, comm);
      break;
    case 1:
      MPI_Ibarrier(comm, &req);
      MPI_Recv(buf, count, dtype, 0, tag, comm, MPI_STATUS_IGNORE);
      MPI_Wait(&req, MPI_STATUS_IGNORE);
      break;
}
```

The MPI library must progress the barrier in the MPI_Recv call. Thus, the MPI_Wait call in rank 0 will eventually complete, which enables the matching MPI_Send so all calls eventually return.

**Example 5.32**

Blocking and nonblocking collective operations do not match. The following example is erroneous.

```
MPI_Request req;

switch(rank) {
    case 0:
      /* erroneous false matching of Alltoall and Ialltoall */
      MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);
      MPI_Wait(&req, MPI_STATUS_IGNORE);
      break;
    case 1:
      /* erroneous false matching of Alltoall and Ialltoall */
      MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);
      break;
}
```

**Example 5.33**

Collective and point-to-point requests can be mixed in functions that enable multiple completions. If started with two processes, the following program is valid.

```
MPI_Request reqs[2];

switch(rank) {
    case 0:
      MPI_Ibarrier(comm, &reqs[0]);
      MPI_Send(buf, count, dtype, 1, tag, comm);
      MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
      break;
    case 1:
      MPI_Irecv(buf, count, dtype, 0, tag, comm, &reqs[0]);
      MPI_Ibarrier(comm, &reqs[1]);
      MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
      break;
}
```

The MPI_Waitall call returns only after the barrier and the receive completed.

**Example 5.34**

Multiple nonblocking collective operations can be outstanding on a single communicator and match in order.

```
MPI_Request reqs[3];

compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
compute(buf3);
MPI_Ibcast(buf3, count, type, 0, comm, &reqs[2]);
MPI_Waitall(3, reqs, MPI_STATUSES_IGNORE);
```

> *Advice to users.* Pipelining and double-buffering techniques can efficiently be used to overlap computation and communication. However, having too many outstanding requests might have a negative impact on performance. (*End of advice to users.*)

> *Advice to implementors.* The use of pipelining may generate many outstanding requests. A high-quality hardware-supported implementation with limited resources should be able to fall back to a software implementation if its resources are exhausted. In this way, the implementation could limit the number of outstanding requests only by the available memory. (*End of advice to implementors.*)

**Example 5.35**

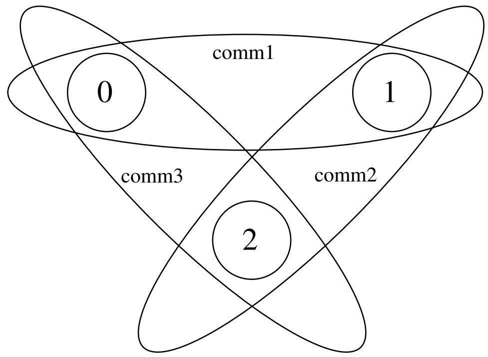


Figure 5.13: Example with overlapping communicators.

Nonblocking collective operations can also be used to enable simultaneous collective operations on multiple overlapping communicators (see Figure 5.13). The following example is started with three processes and three communicators. The first communicator `comm1` includes ranks 0 and 1, `comm2` includes ranks 1 and 2, and `comm3` spans ranks 0 and 2. It is not possible to perform a blocking collective operation on all communicators because there exists no deadlock-free order to invoke them. However, nonblocking collective operations can easily be used to achieve this task.

```
MPI_Request reqs[2];

switch(rank) {
    case 0:
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
      break;
    case 1:
      MPI_Iallreduce(sbuf1, rbuf1, count, dtype, MPI_SUM, comm1, &reqs[0]);
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[1]);
      break;
    case 2:
      MPI_Iallreduce(sbuf2, rbuf2, count, dtype, MPI_SUM, comm2, &reqs[0]);
      MPI_Iallreduce(sbuf3, rbuf3, count, dtype, MPI_SUM, comm3, &reqs[1]);
      break;
}
MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);
```

*Advice to users.* This method can be useful if overlapping neighboring regions (halo or ghost zones) are used in collective operations. The sequence of the two calls in each process is irrelevant because the two nonblocking operations are performed on different communicators. (*End of advice to users.*)

**Example 5.36**

The progress of multiple outstanding nonblocking collective operations is completely independent.

```
MPI_Request reqs[2];

compute(buf1);
MPI_Ibcast(buf1, count, type, 0, comm, &reqs[0]);
compute(buf2);
MPI_Ibcast(buf2, count, type, 0, comm, &reqs[1]);
MPI_Wait(&reqs[1], MPI_STATUS_IGNORE);
/* nothing is known about the status of the first bcast here */
MPI_Wait(&reqs[0], MPI_STATUS_IGNORE);
```

Finishing the second MPI_IBCAST is completely independent of the first one. This means that it is not guaranteed that the first broadcast operation is finished or even started after the second one is completed via `reqs[1]`.

# Chapter 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 [54] 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**.

**Communicators** (see [21, 52, 56]) 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 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.

- 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 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.

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 | group (handle) |
| OUT | size | number of processes in the group (non-negative integer) |

**C binding**

```
int MPI_Group_size(MPI_Group group, int *size)
```

**F08 binding**

```
MPI_Group_size(group, size, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
    INTEGER GROUP, SIZE, IERROR
```

MPI_GROUP_RANK(group, rank)

| | | |
|---|---|---|
| IN | group | group (handle) |
| OUT | rank | rank of the calling process in group, or MPI_UNDEFINED if the process is not a member (non-negative integer) |

**C binding**

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

**F08 binding**

```
MPI_Group_rank(group, rank, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GROUP_RANK(GROUP, RANK, IERROR)
    INTEGER GROUP, RANK, IERROR
```

MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)

| | | |
|---|---|---|
| IN | group1 | group1 (handle) |
| IN | n | number of ranks in ranks1 and ranks2 arrays (integer) |
| IN | ranks1 | array of zero or more valid ranks in group1 (array of non-negative integers) |
| IN | group2 | group2 (handle) |
| OUT | ranks2 | array of corresponding ranks in group2, MPI_UNDEFINED when no correspondence exists. (array of non-negative integers) |

**C binding**

```
int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],
             MPI_Group group2, int ranks2[])
```

**F08 binding**

```
MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    INTEGER, INTENT(IN) ::  n, ranks1(n)
    INTEGER, INTENT(OUT) ::  ranks2(n)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2, IERROR)
    INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR
```

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 (handle) |
| IN | group2 | second group (handle) |
| OUT | result | result (integer) |

**C binding**

```
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
```

**F08 binding**

```
MPI_Group_compare(group1, group2, result, ierror)
```

```
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    INTEGER, INTENT(OUT) :: result
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
    INTEGER GROUP1, GROUP2, RESULT, IERROR
```

MPI_IDENT 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 duplicator. A group, once created, can have several references to it by making copies of the handle. The following constructors address the need for subsets and supersets of existing groups. (*End of rationale.*)

*Advice to implementors.* Each group constructor behaves as if it returned a new group object. When this new group is a copy of an existing group, then one can avoid creating such new objects, using a reference-count mechanism. (*End of advice to implementors.*)

MPI_COMM_GROUP(comm, group)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | group | group corresponding to comm (handle) |

**C binding**

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

**F08 binding**

```
MPI_Comm_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

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) |

**C binding**

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)
```

**F08 binding**

```
MPI_Group_union(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

| | | |
|---|---|---|
| IN | group1 | first group (handle) |
| IN | group2 | second group (handle) |
| OUT | newgroup | intersection group (handle) |

**C binding**

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)
```

**F08 binding**

```
MPI_Group_intersection(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

| | | |
|---|---|---|
| IN | group1 | first group (handle) |
| IN | group2 | second group (handle) |
| OUT | newgroup | difference group (handle) |

**C binding**

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
             MPI_Group *newgroup)
```

**F08 binding**

```
MPI_Group_difference(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
    INTEGER GROUP1, GROUP2, NEWGROUP, IERROR
```

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 the first group.

**intersect** all elements of the first group that are also in the second group, ordered as in the first group.

**difference** all elements of the first group that are not in the second group, ordered as in the 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.

MPI_GROUP_INCL(group, n, ranks, newgroup)

| | | |
|---|---|---|
| IN | group | 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 non-negative integers) |
| OUT | newgroup | new group derived from above, in the order defined by ranks (handle) |

**C binding**

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[],
             MPI_Group *newgroup)
```

**F08 binding**

```
MPI_Group_incl(group, n, ranks, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: n, ranks(n)
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function MPI_GROUP_INCL creates a group newgroup that consists of the n processes in group with ranks ranks[0],..., ranks[n-1]; the process with rank i in newgroup is the process with rank ranks[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.

MPI_GROUP_EXCL(group, n, ranks, newgroup)

| | | |
|---|---|---|
| IN | group | group (handle) |
| IN | n | number of elements in array ranks (integer) |
| IN | ranks | ranks of processes in group not to appear in newgroup (array of non-negative integers) |
| OUT | newgroup | new group derived from above, preserving the order defined by group (handle) |

**C binding**

```
int MPI_Group_excl(MPI_Group group, int n, const int ranks[],
             MPI_Group *newgroup)
```

**F08 binding**

```
MPI_Group_excl(group, n, ranks, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: n, ranks(n)
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
```

The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0] ,... ranks[n-1]. The ordering of processes in newgroup is identical to the ordering in group. Each of the n elements of ranks must be a valid rank in group and all elements must be distinct; otherwise, the program is erroneous. If n = 0, then newgroup is identical to group.

MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)

| | | |
|---|---|---|
| IN | group | group (handle) |
| IN | n | number of triplets in array ranges (integer) |
| IN | ranges | a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be included in newgroup (array of non-negative integers) |
| OUT | newgroup | new group derived from above, in the order defined by ranges (handle) |

**C binding**

```
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)
```

**F08 binding**

```
MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranges(3,n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GROUP_RANGE_INCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

If ranges consists of the triplets

$$(first_1, last_1, stride_1), \ldots, (first_n, last_n, stride_n)$$

then newgroup consists of the sequence of processes in group with ranks

$$first_1, first_1 + stride_1, \ldots, first_1 + \left\lfloor \frac{last_1 - first_1}{stride_1} \right\rfloor stride_1, \ldots,$$

$$first_n, first_n + stride_n, \ldots, first_n + \left\lfloor \frac{last_n - first_n}{stride_n} \right\rfloor stride_n.$$

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 $first_i > last_i$, and $stride_i$ may be negative, but cannot be zero.

The functionality of this routine is specified to be 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_GROUP_INCL. A call to MPI_GROUP_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 | group (handle) |
| IN | n | number of triplets in array ranges (integer) |
| IN | ranges | a one-dimensional array of integer triplets, of the form (first rank, last rank, stride) indicating ranks in group of processes to be excluded from the output group newgroup (array of non-negative integers) |
| OUT | newgroup | new group derived from above, preserving the order in group (handle) |

**C binding**

```
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3],
             MPI_Group *newgroup)
```

**F08 binding**

```
MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranges(3,n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GROUP_RANGE_EXCL(GROUP, N, RANGES, NEWGROUP, IERROR)
    INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR
```

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 whenenever possible, as high-quality implementations will take advantage of this fact. (*End of advice to users.*)

*Advice to implementors.* The range operations should be implemented, if possible, without enumerating the group members, in order to obtain better scalability (time and space). (*End of advice to implementors.*)

### 6.3.3 Group Destructors

MPI_GROUP_FREE(group)

| | | |
|---|---|---|
| INOUT | group | group (handle) |

**C binding**

```
int MPI_Group_free(MPI_Group *group)
```

**F08 binding**

```
MPI_Group_free(group, ierror)
    TYPE(MPI_Group), INTENT(INOUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GROUP_FREE(GROUP, IERROR)
    INTEGER GROUP, IERROR
```

This operation marks a group object for deallocation. The handle group is set to MPI_GROUP_NULL by the call. Any on-going operation using this group will complete normally.

> *Advice to implementors.* One can keep a reference count that is incremented for each call to MPI_COMM_GROUP, MPI_COMM_CREATE, MPI_COMM_DUP, and MPI_COMM_IDUP, 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.

MPI_COMM_SIZE(comm, size)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | size | number of processes in the group of comm (non-negative integer) |

**C binding**

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

**F08 binding**

```
MPI_Comm_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

*Rationale.* This function is equivalent to accessing the communicator's group with MPI_COMM_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 unless the number of processes has been changed by using the functions described in Chapter 10; note that the number of processes in MPI_COMM_WORLD does not change during the life of an MPI program.

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 $0 \ldots$size$-1$, where size is the return value of MPI_COMM_SIZE.(*End of advice to users.*)

MPI_COMM_RANK(comm, rank)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | rank | rank of the calling process in group comm (non-negative integer) |

**C binding**

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

**F08 binding**

```
MPI_Comm_rank(comm, rank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: rank
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_RANK(COMM, RANK, IERROR)
    INTEGER COMM, RANK, IERROR
```

> *Rationale.* This function is equivalent to accessing the communicator's group with MPI_COMM_GROUP (see above), computing the rank using MPI_GROUP_RANK, 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 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.*)

MPI_COMM_COMPARE(comm1, comm2, result)

| | | |
|---|---|---|
| IN | comm1 | first communicator (handle) |
| IN | comm2 | second communucator (handle) |
| OUT | result | result (integer) |

**C binding**

```
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
```

**F08 binding**

```
MPI_Comm_compare(comm1, comm2, result, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm1, comm2
    INTEGER, INTENT(OUT) :: result
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
    INTEGER COMM1, COMM2, RESULT, IERROR
```

MPI_IDENT results if and only if comm1 and comm2 are handles for the same object (identical groups and same contexts). MPI_CONGRUENT results if the underlying groups are identical in constituents and rank order; these communicators differ only by context. MPI_SIMILAR results if the group members of both communicators are the same but the rank order differs. MPI_UNEQUAL results otherwise.

### 6.4.2 Communicator Constructors

The following are collective functions that are invoked by all processes in the group or groups associated with comm, with the exception of MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.

*Rationale.* Note that there is a chicken-and-egg aspect to MPI in that a communicator is needed to create a new communicator. The base communicator for all MPI communicators is predefined outside of MPI, and is MPI_COMM_WORLD. This model was arrived at after considerable debate, and was chosen to increase "safety" of programs written in MPI. (*End of rationale.*)

This chapter presents the following communicator construction routines: MPI_COMM_CREATE, MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO and MPI_COMM_SPLIT can be used to create both intracommunicators and intercommunicators; MPI_COMM_CREATE_GROUP and MPI_INTERCOMM_MERGE (see Section 6.6.2) can be used to create intracommunicators; and MPI_INTERCOMM_CREATE (see Section 6.6.2) can be used to create 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 intercommunicator are called the *left* and *right* groups. A process in an intercommunicator 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 intercommunicator that is not relative to any particular process (as the local and remote groups are).

MPI_COMM_DUP(comm, newcomm)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | newcomm | copy of comm (handle) |

**C binding**

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Comm_dup(comm, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

MPI_COMM_DUP duplicates the existing communicator comm with associated key values and topology information. 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 communicator. MPI_COMM_DUP returns in newcomm a new communicator with the same group or groups, same topology, and any copied cached information, but a new context (see Section 6.7.1).

*Advice to users.* This operation is used to provide a parallel library with a duplicate communication 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 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_DUP_WITH_INFO(comm, info, newcomm)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| IN | info | info object (handle) |
| OUT | newcomm | copy of comm (handle) |

**C binding**

```
int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
    INTEGER COMM, INFO, NEWCOMM, IERROR
```

MPI_COMM_DUP_WITH_INFO behaves exactly as MPI_COMM_DUP except that the hints provided by the argument info are associated with the output communicator newcomm.

*Rationale.* It is expected that some hints will only be valid at communicator creation time. However, for legacy reasons, most communicator creation calls do not provide an info argument. One may associate info hints with a duplicate of any communicator at creation time through a call to MPI_COMM_DUP_WITH_INFO. (*End of rationale.*)

MPI_COMM_IDUP(comm, newcomm, request)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | newcomm | copy of comm (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)
```

**F08 binding**

```
MPI_Comm_idup(comm, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
    INTEGER COMM, NEWCOMM, REQUEST, IERROR
```

MPI_COMM_IDUP is a nonblocking variant of MPI_COMM_DUP. With the exception of its nonblocking behavior, the semantics of MPI_COMM_IDUP are as if MPI_COMM_DUP was executed at the time that MPI_COMM_IDUP is called. For example, attributes changed after MPI_COMM_IDUP will not be copied to the new communicator. All restrictions and assumptions for nonblocking collective operations (see Section 5.12) apply to MPI_COMM_IDUP and the returned request.

It is erroneous to use the communicator newcomm as an input argument to other MPI functions before the MPI_COMM_IDUP operation completes.

MPI_COMM_IDUP_WITH_INFO(comm, info, newcomm, request)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| IN | info | info object (handle) |
| OUT | newcomm | copy of comm (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,
              MPI_Comm *newcomm, MPI_Request *request)
```

**F08 binding**

```
MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)
    INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR
```

MPI_COMM_IDUP_WITH_INFO is a nonblocking variant of MPI_COMM_DUP_WITH_INFO. With the exception of its nonblocking behavior, the semantics of MPI_COMM_IDUP_WITH_INFO are as if MPI_COMM_DUP_WITH_INFO was executed at the time that MPI_COMM_IDUP_WITH_INFO is called. For example, attributes or info hints changed after MPI_COMM_IDUP_WITH_INFO will not be copied to the new communicator. All restrictions and assumptions for nonblocking collective operations (see Section 5.12) apply to MPI_COMM_IDUP_WITH_INFO and the returned request.

It is erroneous to use the communicator newcomm as an input argument to other MPI functions before the MPI_COMM_IDUP_WITH_INFO operation completes.

> *Rationale.* The MPI_COMM_IDUP and MPI_COMM_IDUP_WITH_INFO functions are crucial for the development of purely nonblocking libraries (see [36]). (*End of rationale.*)

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) |

**C binding**

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Comm_create(comm, group, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(IN) ::  group
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

If comm is an intracommunicator, this function returns a new communicator newcomm with communication group defined by the group argument. No cached information propagates from comm to newcomm. Each process must call MPI_COMM_CREATE with a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. The processes may specify different values for the group argument. If a process calls with a non-empty group then all processes in that group must call the function with the same group as argument, that is the same processes in the same order. Otherwise, the call is erroneous. This implies that the set of groups specified across the processes must be disjoint. If the calling process is a member of the group given as group argument, then newcomm is a communicator with group as its associated group. In the case that a process calls with a group to which it does not belong, e.g., MPI_GROUP_EMPTY, then MPI_COMM_NULL is returned as newcomm. The function is collective and must be called by all processes in the group of comm.

> *Rationale.* The interface supports the original mechanism from MPI-1.1, which required the same group in all processes of comm. It was extended in MPI-2.2 to allow the use of disjoint subgroups in order to allow implementations to eliminate unnecessary communication that MPI_COMM_SPLIT would incur when the user already knows the membership of the disjoint subgroups. (*End of rationale.*)

*Rationale.* The requirement that the entire group of comm participate in the call stems from the following considerations:

- It allows the implementation to layer MPI_COMM_CREATE on top of regular collective communications.
- It provides additional safety, in particular in the case where partially overlapping groups are used to create new communicators.
- It permits implementations to sometimes avoid communication related to context creation.

(*End of rationale.*)

*Advice to users.* MPI_COMM_CREATE provides a means to subset a group of processes for the purpose of separate MIMD computation, with separate communication space. newcomm, which emerges from MPI_COMM_CREATE, can be used in subsequent calls to MPI_COMM_CREATE (or other communicator constructors) to further subdivide a computation into parallel sub-computations. A more general service is provided by MPI_COMM_SPLIT, below. (*End of advice to users.*)

*Advice to implementors.* When calling MPI_COMM_DUP, all processes call with the same group (the group associated with the communicator). When calling MPI_COMM_CREATE, the processes provide the same group or disjoint subgroups. For both calls, it is theoretically possible to agree on a group-wide unique context with no communication. However, local execution of these functions requires use of a larger context name space and reduces error checking. Implementations may strike various compromises between these conflicting goals, such as bulk allocation of multiple contexts in one collective operation.

Important: If new communicators are created without synchronizing the processes involved then the communication system must be able to cope with messages arriving in a context that has not yet been allocated at the receiving process. (*End of advice to implementors.*)

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. All processes in the same local group of comm must specify the same value for group, i.e., the same members in the same order. 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.*)

**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.

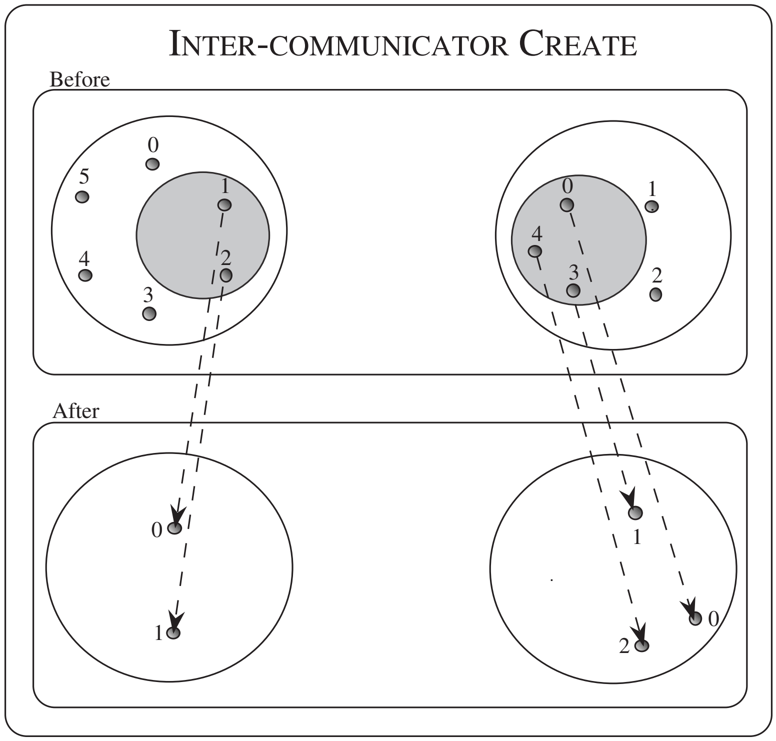


Figure 6.1: Intercommunicator creation using MPI_COMM_CREATE extended to intercommunicators. The input groups are those in the grey circle.

```
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);
```

MPI_COMM_CREATE_GROUP(comm, group, tag, newcomm)

| | | |
|---|---|---|
| IN | comm | intracommunicator (handle) |
| IN | group | group, which is a subset of the group of comm (handle) |
| IN | tag | tag (integer) |
| OUT | newcomm | new communicator (handle) |

**C binding**

```
int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
             MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  tag
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_CREATE_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR
```

MPI_COMM_CREATE_GROUP is similar to MPI_COMM_CREATE; however, MPI_COMM_CREATE must be called by all processes in the group of comm, whereas MPI_COMM_CREATE_GROUP must be called by all processes in group, which is a subgroup of the group of comm. In addition, MPI_COMM_CREATE_GROUP requires that comm is an intracommunicator. MPI_COMM_CREATE_GROUP returns a new intracommunicator, newcomm, for which the group argument defines the communication group. No cached information propagates from comm to newcomm. Each process must provide a group argument that is a subgroup of the group associated with comm; this could be MPI_GROUP_EMPTY. If a non-empty group is specified, then all processes in that group must call the function, and each of these processes must provide the same arguments, including a group that contains the same members with the same ordering. Otherwise the call is erroneous. If the calling process is a member of the group given as the group argument, then newcomm is a communicator with group as its associated group. If the calling process is not a member of group, e.g., group is MPI_GROUP_EMPTY, then the call is a local operation and MPI_COMM_NULL is returned as newcomm.

*Rationale.* Functionality similar to MPI_COMM_CREATE_GROUP can be implemented through repeated MPI_INTERCOMM_CREATE and MPI_INTERCOMM_MERGE calls that start with the MPI_COMM_SELF communicators at each process in group and build up an intracommunicator with group group [16]. Such an algorithm requires the creation of many intermediate communicators; MPI_COMM_CREATE_GROUP can provide a more efficient implementation that avoids this overhead. (*End of rationale.*)

*Advice to users.* An intercommunicator can be created collectively over processes in the union of the local and remote groups by creating the local communicator using

MPI_COMM_CREATE_GROUP and using that communicator as the local communicator argument to MPI_INTERCOMM_CREATE. (*End of advice to users.*)

The tag argument does not conflict with tags used in point-to-point communication and is not permitted to be a wildcard. If multiple threads at a given process perform concurrent MPI_COMM_CREATE_GROUP operations, the user must distinguish these operations by providing different tag or comm arguments.

*Advice to users.* MPI_COMM_CREATE may provide lower overhead than MPI_COMM_CREATE_GROUP because it can take advantage of collective communication on comm when constructing newcomm. (*End of advice to users.*)

MPI_COMM_SPLIT(comm, color, key, newcomm)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| IN | color | control of subset assignment (integer) |
| IN | key | control of rank assignment (integer) |
| OUT | newcomm | new communicator (handle) |

**C binding**

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Comm_split(comm, color, key, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  color, key
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument key, with ties broken according to their rank in the old group. A new communicator is created for each subgroup and returned in newcomm. A process may supply the color value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL. This is a collective call, but each process is permitted to provide different values for color and key.

With an intracommunicator comm, a call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where processes that are members of their group argument provide color = number of the group (based on a unique numbering of all disjoint groups) and key = rank in group, and all processes that are not members of their group argument provide color = MPI_UNDEFINED.

The value of color must be non-negative or MPI_UNDEFINED.

*Advice to users.* This is an extremely powerful mechanism for dividing a single communicating group of processes into $k$ subgroups, with $k$ chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communicator will be non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra. For intracommunicators, MPI_COMM_SPLIT provides similar capability as MPI_COMM_CREATE to split a communicating group into disjoint subgroups. MPI_COMM_SPLIT is useful when some processes do not have complete information of the other members in their group, but all processes know (the color of) the group to which they belong. In this case, the MPI implementation discovers the other group members via communication. MPI_COMM_CREATE is useful when all processes have complete information of the members of their group. In this case, MPI can avoid the extra communication required to discover group membership. MPI_COMM_CREATE_GROUP is useful when all processes in a given group have complete information of the members of their group and synchronization with processes outside the group can be avoided.

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 does not really care about the rank-order of the processes in the new communicator. (*End of advice to users.*)

*Rationale.* color is restricted to be non-negative, so as not to confict 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.

*Advice to users.* For intercommunicators, MPI_COMM_SPLIT is more general than MPI_COMM_CREATE. A single call to MPI_COMM_SPLIT can create a set of disjoint intercommunicators, while a call to MPI_COMM_CREATE creates only one. (*End of advice to users.*)

**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.

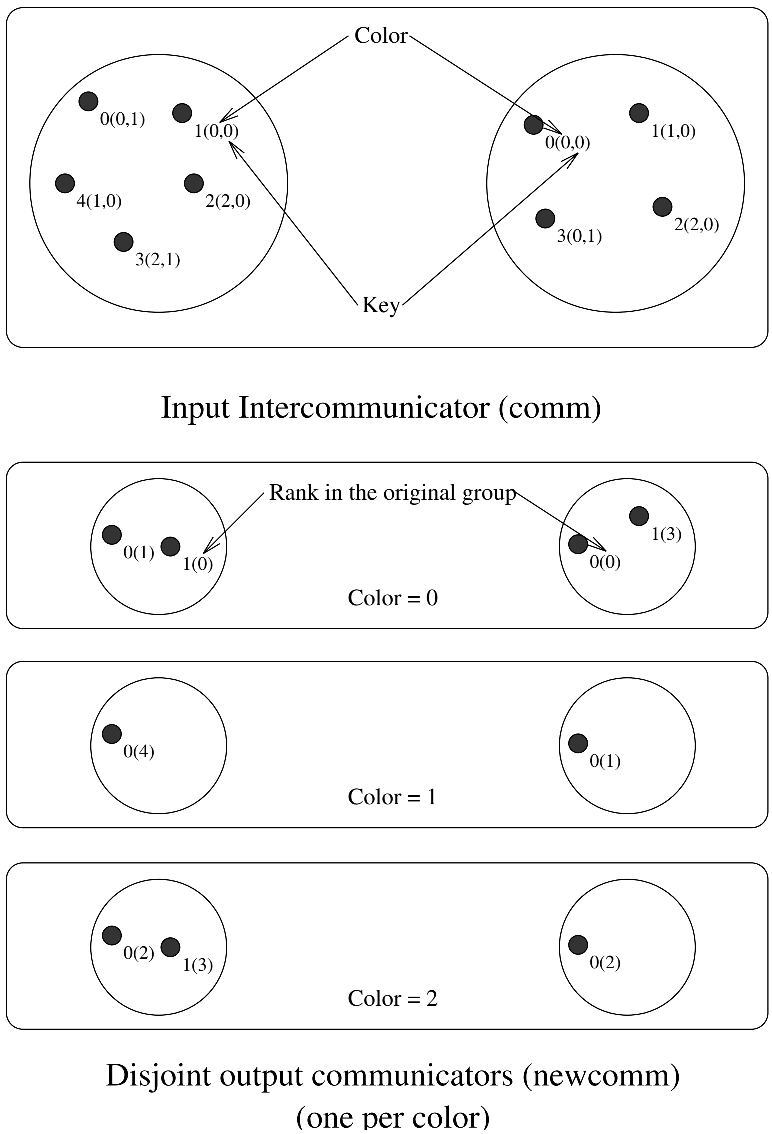


Figure 6.2: Intercommunicator construction achieved by splitting an existing intercommunicator with MPI_COMM_SPLIT extended to intercommunicators.

```
/* 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 */
...

/* 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);
```

MPI_COMM_SPLIT_TYPE(comm, split_type, key, info, newcomm)

| | | | |
|---|---|---|---|
| IN | comm | communicator (handle) |
| IN | split_type | type of processes to be grouped together (integer) |
| IN | key | control of rank assignment (integer) |
| IN | info | info argument (handle) |
| OUT | newcomm | new communicator (handle) |

**C binding**

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type, int key,
             MPI_Info info, MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  split_type, key
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
    INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
```

This function partitions the group associated with comm into disjoint subgroups, based on the type specified by split_type. Each subgroup contains all processes of the same type. 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. This is a collective call; all processes must provide the same split_type, but each process is permitted to provide different values for key. An exception to this rule is that a process may supply the type value MPI_UNDEFINED, in which case newcomm returns MPI_COMM_NULL.

The following type is predefined by MPI:

MPI_COMM_TYPE_SHARED — this type splits the communicator into subcommunicators, each of which can create a shared memory region.

*Advice to implementors.* Implementations can define their own types, or use the info argument, to assist in creating communicators that help expose platform-specific information to the application. (*End of advice to implementors.*)

### 6.4.3 Communicator Destructors

MPI_COMM_FREE(comm)

| | | |
|---|---|---|
| INOUT | comm | communicator to be destroyed (handle) |

**C binding**

```
int MPI_Comm_free(MPI_Comm *comm)
```

**F08 binding**

```
MPI_Comm_free(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_FREE(COMM, IERROR)
    INTEGER COMM, IERROR
```

This collective operation marks the communication object for deallocation. The handle is set to MPI_COMM_NULL. Any pending operations that use this communicator will complete normally; the object is actually deallocated only if there are no other active references to it. This call applies to intra- and inter-communicators. The delete callback functions for all cached attributes (see Section 6.7) are called in arbitrary order.

*Advice to implementors.* Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

### 6.4.4 Communicator Info

Hints specified via info (see Chapter 9) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or minimize use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_COMM_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per communicator basis, in MPI_COMM_DUP_WITH_INFO, MPI_COMM_IDUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_SPLIT_TYPE, MPI_DIST_GRAPH_CREATE, and MPI_DIST_GRAPH_CREATE_ADJACENT, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_COMM_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

*Advice to implementors.* It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. Needless to say, no hint can be mandatory. However, for each hint used by a specific implementation, a default value must be provided when the user does not specify a value for this hint. (*End of advice to implementors.*)

Info hints are not propagated by MPI from one communicator to another. The following info keys are valid for all communicators.

mpi_assert_no_any_tag **(boolean, default:** false**):** If set to true, then the implementation may assume that the process will not use the MPI_ANY_TAG wildcard on the given communicator.

mpi_assert_no_any_source **(boolean, default:** false**):** If set to true, then the implementation may assume that the process will not use the MPI_ANY_SOURCE wildcard on the given communicator.

mpi_assert_exact_length **(boolean, default:** false**):** If set to true, then the implementation may assume that the lengths of messages received by the process are equal to the lengths of the corresponding receive buffers, for point-to-point communication operations on the given communicator.

mpi_assert_allow_overtaking **(boolean, default:** false**):** If set to true, then the implementation may assume that point-to-point communications on the given communicator do not rely on the non-overtaking rule specified in Section 3.5. In other words, the application asserts that send operations are not required to be matched at the receiver in the order in which the send operations were posted by the sender, and receive operations are not required to be matched in the order in which they were posted by the receiver.

*Advice to users.* Use of the mpi_assert_allow_overtaking info key can result in nondeterminism in the message matching order. (*End of advice to users.*)

*Advice to users.* Some optimizations may only be possible when all processes in the group of the communicator provide a given info key with the same value. (*End of advice to users.*)

MPI_COMM_SET_INFO(comm, info)

| | | |
|---|---|---|
| INOUT | comm | communicator (handle) |
| IN | info | info object (handle) |

**C binding**

```
int MPI_Comm_set_info(MPI_Comm comm, MPI_Info info)
```

**F08 binding**

```
MPI_Comm_set_info(comm, info, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_SET_INFO(COMM, INFO, IERROR)
    INTEGER COMM, INFO, IERROR
```

MPI_COMM_SET_INFO updates the hints of the communicator associated with comm using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_COMM_SET_INFO. MPI_COMM_SET_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

*Advice to users.* Some info items that an implementation can use when it creates a communicator cannot easily be changed once the communicator has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_COMM_SET_INFO. MPI_COMM_GET_INFO can be used to determine whether updates to existing info hints were ignored by the implementation. (*End of advice to users.*)

*Advice to users.* Setting info hints on the predefined communicators MPI_COMM_WORLD and MPI_COMM_SELF may have unintended effects, as changes to these global objects may affect all components of the application, including libraries and tools. Users must ensure that all components of the application that use a given communicator, including libraries and tools, can comply with any info hints associated with that communicator. (*End of advice to users.*)

MPI_COMM_GET_INFO(comm, info_used)

| | | |
|---|---|---|
| IN | comm | communicator object (handle) |
| OUT | info_used | new info object (handle) |

**C binding**

```
int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)
```

**F08 binding**

```
MPI_Comm_get_info(comm, info_used, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(OUT) ::  info_used
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_GET_INFO(COMM, INFO_USED, IERROR)
    INTEGER COMM, INFO_USED, IERROR
```

MPI_COMM_GET_INFO returns a new info object containing the hints of the communicator associated with comm. The current setting of all hints related to this communicator is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

## 6.5 Motivating Examples

### 6.5.1 Current Practice #1

Example #1a:

```
int main(int argc, char *argv[])
{
  int me, size;
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  (void)printf("Process %d size %d\n", me, size);
  ...
```

```
  MPI_Finalize();
  return 0;
}
```

Example #1a is a do-nothing program that initializes itself, and refers to the "all" communicator, and prints a message. It terminates itself too. This example does not imply that MPI supports `printf`-like communication itself.
Example #1b (supposing that `size` is even):

```
int main(int argc, char *argv[])
{
   int me, size;
   int SOME_TAG = 0;
   ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

   if((me % 2) == 0)
   {
      /* send unless highest-numbered process */
      if((me + 1) < size)
         MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
   }
   else
      MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);

   ...
   MPI_Finalize();
   return 0;
}
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

### 6.5.2 Current Practice #2

```
int main(int argc, char *argv[])
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
```

```
        /* get input, create buffer ``data'' */
        ...
    }

    MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

    ...
    MPI_Finalize();
    return 0;
  }
```

This example illustrates the use of a collective communication.

### 6.5.3 (Approximate) Current Practice #3

```
int main(int argc, char *argv[])
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group group_world, grprem;
  MPI_Comm commslave;
  static int ranks[] = {0};
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* local */

  MPI_Group_excl(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_buf,count, MPI_INT, MPI_SUM, 1, commslave);
    ...
    MPI_Comm_free(&commslave);
  }
  /* zero falls through immediately to this reduce, others do later... */
  MPI_Reduce(send_buf2, recv_buf2, count2,
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Group_free(&group_world);
  MPI_Group_free(&grprem);
  MPI_Finalize();
  return 0;
}
```

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

int main(int argc, char *argv[])
{
  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group group_world, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &group_world);

  MPI_Group_incl(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(&group_world);
```

```
  MPI_Group_free(&subgroup);
  MPI_Finalize();
  return 0;
}
```

### 6.5.5 Library Example #1

The main program:

```
int 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);
     ...
     /* see if done */
     ...
  }
  user_end_op(libh_a);
  user_end_op(libh_b);

  uninit_user_lib(libh_a);
  uninit_user_lib(libh_b);
  MPI_Finalize();
  return 0;
}
```

The user library initialization code:

```
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:

```
  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:

```
  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:

```
  void uninit_user_lib(user_lib_t *handle)
  {
    MPI_Comm_free(&(handle->comm));
    free(handle);
  }
```

### 6.5.6 Library Example #2

The main program:

```
  int main(int argc, char *argv[])
  {
    int ma, mb;
    MPI_Group 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, &group_world);

  MPI_Group_incl(group_world, size_list_a, list_a, &group_a);
  MPI_Group_incl(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);
    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);
  MPI_Group_free(&group_world);
  MPI_Finalize();
  return 0;
}
```

The library:

```
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);
        ...
     }
  else
```

```
      {
        /* work */
        MPI_Send(..., 0, ARBITRARY_TAG, comm);
        ...
      }
#ifdef EXAMPLE_2C
      /* include (resp, exclude) for safety (resp, no safety): */
      MPI_Barrier(comm);
#endif
    }
```

The above example is really three examples, depending on whether or not one includes rank 3 in list_b, and whether or not a synchronize is included in lib_call. This example illustrates that, despite contexts, subsequent calls to lib_call with the same context need not be safe from one another (colloquially, "back-masking"). Safety is realized if the MPI_Barrier is added. What this demonstrates is that libraries have to be written carefully, even with contexts. When rank 3 is excluded, then the synchronize is not needed to get safety from back-masking.

Algorithms like "reduce" and "allreduce" have strong enough source selectivity properties so that they are inherently okay (no back-masking), provided that MPI provides basic guarantees. So are multiple calls to a typical tree-broadcast algorithm with the same root or different roots (see [56]). 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 back-masking 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 except for MPI_COMM_IDUP and require that the local and remote groups be disjoint.

*Advice to users.* The groups must be disjoint for several reasons. Primarily, this is the intent of the 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 intracommunicator; this ranking makes little sense if the groups are not disjoint. In addition, the natural extension of collective operations to inter-communicators makes the most sense when the groups are disjoint. (*End of advice to 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_CART_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 $\mathcal{P}$, which has an inter-communicator $\mathbf{C}_{\mathcal{P}}$, and a process **Q** in group $\mathcal{Q}$, which has an inter-communicator $\mathbf{C}_{\mathcal{Q}}$. Then

- $\mathbf{C}_{\mathcal{P}}$**.group** describes the group $\mathcal{Q}$ and $\mathbf{C}_{\mathcal{Q}}$**.group** describes the group $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.send_context** = $\mathbf{C}_{\mathcal{Q}}$**.receive_context** and the context is unique in $\mathcal{Q}$; $\mathbf{C}_{\mathcal{P}}$**.receive_context** = $\mathbf{C}_{\mathcal{Q}}$**.send_context** and this context is unique in $\mathcal{P}$.
- $\mathbf{C}_{\mathcal{P}}$**.source** is rank of **P** in $\mathcal{P}$ and $\mathbf{C}_{\mathcal{Q}}$**.source** is rank of **Q** in $\mathcal{Q}$.

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(comm, flag)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | flag | (logical) |

**C binding**

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

**F08 binding**

```
MPI_Comm_test_inter(comm, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
```

```
    LOGICAL FLAG
```

This local routine allows the calling process to determine if a communicator is an 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.

| | |
|---|---|
| MPI_COMM_SIZE | returns the size of the local group. |
| MPI_COMM_GROUP | returns the local group. |
| MPI_COMM_RANK | returns the rank in the local group |

Table 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 or MPI_SIMILAR. In particular, it is possible for MPI_SIMILAR to result because either the local or remote groups were similar but not identical.

The following accessors provide consistent access to the remote group of an inter-communicator. The following are all local operations.

MPI_COMM_REMOTE_SIZE(comm, size)

| | | |
|---|---|---|
| IN | comm | inter-communicator (handle) |
| OUT | size | number of processes in the remote group of comm (non-negative integer) |

**C binding**

```
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
```

**F08 binding**

```
MPI_Comm_remote_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_REMOTE_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR
```

MPI_COMM_REMOTE_GROUP(comm, group)

| | | |
|---|---|---|
| IN | comm | inter-communicator (handle) |
| OUT | group | remote group corresponding to comm (handle) |

**C binding**

```
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
```

**F08 binding**

```
MPI_Comm_remote_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_REMOTE_GROUP(COMM, GROUP, IERROR)
    INTEGER COMM, GROUP, IERROR
```

> *Rationale.* Symmetric access to both the local and remote groups of an inter-communicator is important, so this function, as well as MPI_COMM_REMOTE_SIZE have been provided. (*End of rationale.*)

### 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.

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 (non-negative 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 (non-negative integer) |
| IN | tag | tag (integer) |
| OUT | newintercomm | new inter-communicator (handle) |

**C binding**

```
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader,
             MPI_Comm peer_comm, int remote_leader, int tag,
             MPI_Comm *newintercomm)
```

**F08 binding**

```
MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
             tag, newintercomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  local_comm, peer_comm
    INTEGER, INTENT(IN) ::  local_leader, remote_leader, tag
    TYPE(MPI_Comm), INTENT(OUT) ::  newintercomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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
```

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.

MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)

| | | |
|---|---|---|
| IN | intercomm | Inter-Communicator (handle) |
| IN | high | (logical) |
| OUT | newintracomm | new intra-communicator (handle) |

**C binding**

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high,
             MPI_Comm *newintracomm)
```

**F08 binding**

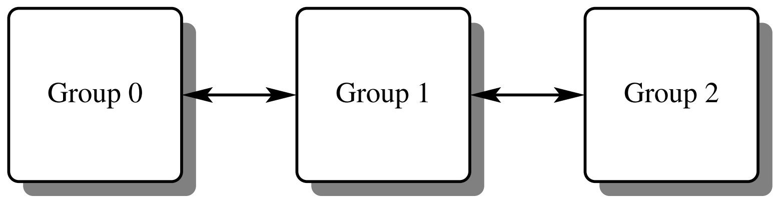


Figure 6.3: Three-group pipeline

```
MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  intercomm
    LOGICAL, INTENT(IN) ::  high
    TYPE(MPI_Comm), INTENT(OUT) ::  newintracomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
    INTEGER INTERCOMM, NEWINTRACOMM, IERROR
    LOGICAL HIGH
```

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the "low" group 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.

```
int 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);
  case 0:
  case 2:
     MPI_Comm_free(&myFirstComm);
     break;
  }

  MPI_Finalize();
  return 0;
}
```

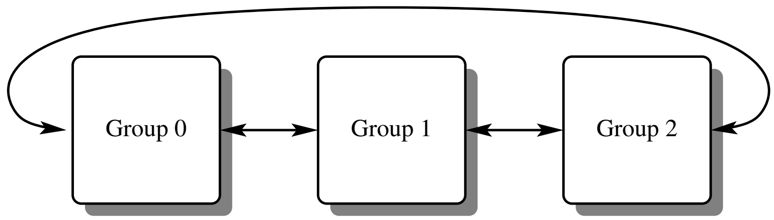


Figure 6.4: Three-group ring

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.

```
int main(int argc, char *argv[])
{
  MPI_Comm   myComm;      /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm; /* inter-communicators */
  MPI_Comm   mySecondComm;
  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. */
    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. */
    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. */
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 0,
                         2, &myFirstComm);
    MPI_Intercomm_create(myComm, 0, MPI_COMM_WORLD, 1,
                         12, &mySecondComm);
  }

  /* Do some work ... */

  /* Then free communicators before terminating... */
  MPI_Comm_free(&myFirstComm);
  MPI_Comm_free(&mySecondComm);
  MPI_Comm_free(&myComm);
  MPI_Finalize();
  return 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 attribute-caching mechanism. (*End of advice to users.*)

*Rationale.* In one extreme one can allow caching on all opaque handles. The other extreme is to only allow it on communicators. Caching has a cost associated with it and should only be allowed when it is clearly needed and the increased cost is modest. This is the reason that windows and datatypes were added but not other handles. (*End of rationale.*)

One difficulty is the potential for size differences between Fortran integers and C pointers. For this reason, the Fortran versions of these routines use integers of kind MPI_ADDRESS_KIND.

*Advice to implementors.* High-quality implementations should raise an error when a keyval that was created by a call to MPI_XXX_CREATE_KEYVAL is used with an object of the wrong type with a call to MPI_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

### 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 or MPI_COMM_IDUP (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 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 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:

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 function |

**C binding**

```
int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
             MPI_Comm_delete_attr_function *comm_delete_attr_fn,
             int *comm_keyval, void *extra_state)
```

**F08 binding**

```
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Comm_copy_attr_function) ::  comm_copy_attr_fn
    PROCEDURE(MPI_Comm_delete_attr_function) ::  comm_delete_attr_fn
    INTEGER, INTENT(OUT) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

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.
The C callback functions are:

```
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

```
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm, 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.
With the `mpi_f08` module, the Fortran callback functions are:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Comm) ::  oldcomm
      INTEGER ::  comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

and

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
  attribute_val, extra_state, ierror)
      TYPE(MPI_Comm) ::  comm
      INTEGER ::  comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

With the `mpi` module and `mpif.h`, the Fortran callback functions are:

```
SUBROUTINE COMM_COPY_ATTR_FUNCTION(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_FUNCTION(COMM, COMM_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The comm_copy_attr_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP or MPI_COMM_IDUP. comm_copy_attr_fn should be of type MPI_Comm_copy_attr_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 or .FALSE., then the attribute is deleted in the duplicated communicator. Otherwise (flag = 1 or .TRUE.), 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 or MPI_COMM_IDUP will fail).

The argument comm_copy_attr_fn may be specified as MPI_COMM_NULL_COPY_FN or MPI_COMM_DUP_FN from either C or Fortran. MPI_COMM_NULL_COPY_FN is a function that does nothing other than returning flag = 0 or .FALSE. (depending on whether the keyval was created with a C or Fortran binding to MPI_COMM_CREATE_KEYVAL) and MPI_SUCCESS. MPI_COMM_DUP_FN is a simple-minded copy function that sets flag = 1 or .TRUE., returns the value of attribute_val_in in attribute_val_out, and returns MPI_SUCCESS.

These replace the MPI-1 predefined callbacks MPI_NULL_COPY_FN and MPI_DUP_FN, whose use is deprecated.

*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 or Fortran. MPI_COMM_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. MPI_COMM_NULL_DELETE_FN replaces MPI_NULL_DELETE_FN, whose use is deprecated.

If an attribute copy function or attribute delete function returns other than MPI_SUCCESS, then the call that caused it to be invoked (for example, MPI_COMM_FREE), is erroneous.

The special key value MPI_KEYVAL_INVALID is never returned by MPI_COMM_CREATE_KEYVAL. Therefore, it can be used for static initialization of key values.

*Advice to implementors.* The predefined Fortran functions MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and MPI_COMM_NULL_DELETE_FN are defined in the mpi module (and mpif.h) and the mpi_f08 module with the same name, but with different interfaces. Each function can coexist twice with the same name in the same MPI library, one routine as an implicit interface outside of the mpi module, i.e., declared as EXTERNAL, and the other routine within mpi_f08 declared with CONTAINS. These routines have different link names, which are also different to the link names used for the routines used in C. (*End of advice to implementors.*)

*Advice to users.* Callbacks, including the predefined Fortran functions MPI_COMM_NULL_COPY_FN, MPI_COMM_DUP_FN, and MPI_COMM_NULL_DELETE_FN should not be passed from one application routine that uses the mpi_f08 module to another application routine that uses the mpi module or mpif.h, and vice versa; see also the advice to users on page 676. (*End of advice to users.*)

MPI_COMM_FREE_KEYVAL(comm_keyval)

| | | |
|---|---|---|
| INOUT | comm_keyval | key value (integer) |

**C binding**

```
int MPI_Comm_free_keyval(int *comm_keyval)
```

**F08 binding**

```
MPI_Comm_free_keyval(comm_keyval, ierror)
    INTEGER, INTENT(INOUT) :: comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_FREE_KEYVAL(COMM_KEYVAL, IERROR)
    INTEGER COMM_KEYVAL, IERROR
```

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 explictly 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.

MPI_COMM_SET_ATTR(comm, comm_keyval, attribute_val)

| | | |
|---|---|---|
| INOUT | comm | communicator to which attribute will be attached (handle) |
| IN | comm_keyval | key value (integer) |
| IN | attribute_val | attribute value |

**C binding**

```
int MPI_Comm_set_attr(MPI_Comm comm, int comm_keyval, void* attribute_val)
```

**F08 binding**

```
MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

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.

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) |

**C binding**

```
int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void* attribute_val,
             int *flag)
```

**F08 binding**

```
MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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
```

Retrieves attribute value by key. The call is erroneous if there is no key with value keyval. On the other hand, the call is correct if the key value exists, but no attribute is attached on comm for that key; in such case, the call returns flag = false. In particular MPI_KEYVAL_INVALID is an erroneous key value.

*Advice to users.* The call to MPI_Comm_set_attr passes in attribute_val the *value* of the attribute; the call to MPI_Comm_get_attr passes in attribute_val the *address* of the location where the attribute value is to be returned. Thus, if the attribute value itself is a pointer of type void*, then the actual attribute_val parameter to MPI_Comm_set_attr will be of type void* and the actual attribute_val parameter to MPI_Comm_get_attr will be of type void**. (*End of advice to users.*)

*Rationale.* The use of a formal parameter attribute_val of type void* (rather than void**) avoids the messy type casting that would be needed if the attribute value is declared with a type other than void*. (*End of rationale.*)

MPI_COMM_DELETE_ATTR(comm, comm_keyval)

| | | |
|---|---|---|
| INOUT | comm | communicator from which the attribute is deleted (handle) |
| IN | comm_keyval | key value (integer) |

**C binding**

```
int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)
```

**F08 binding**

```
MPI_Comm_delete_attr(comm, comm_keyval, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
```

Delete attribute from cache by key. This function invokes the attribute delete function comm_delete_attr_fn specified when the keyval was created. The call will fail if the comm_delete_attr_fn function returns an error code other than MPI_SUCCESS.

Whenever a communicator is replicated using the function MPI_COMM_DUP or MPI_COMM_IDUP, all call-back copy functions for attributes that are currently set are invoked (in arbitrary order). Whenever a communicator is deleted using the function MPI_COMM_FREE all callback delete functions for attributes that are currently set are invoked.

### 6.7.3 Windows

The functions for caching on windows are:

MPI_WIN_CREATE_KEYVAL(win_copy_attr_fn, win_delete_attr_fn, win_keyval, extra_state)

| | | |
|---|---|---|
| IN | win_copy_attr_fn | copy callback function for win_keyval (function) |
| 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 function |

**C binding**

```
int MPI_Win_create_keyval(MPI_Win_copy_attr_function *win_copy_attr_fn,
             MPI_Win_delete_attr_function *win_delete_attr_fn,
             int *win_keyval, void *extra_state)
```

**F08 binding**

```
MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Win_copy_attr_function) :: win_copy_attr_fn
    PROCEDURE(MPI_Win_delete_attr_function) :: win_delete_attr_fn
    INTEGER, INTENT(OUT) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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
```

The argument win_copy_attr_fn may be specified as MPI_WIN_NULL_COPY_FN or MPI_WIN_DUP_FN from either 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 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);
```

and

```
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
             void *attribute_val, void *extra_state);
```

With the mpi_f08 module, the Fortran callback functions are:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Win) :: oldwin
      INTEGER :: win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL :: flag
```

and

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
```

```
  extra_state, ierror)
      TYPE(MPI_Win) :: win
      INTEGER :: win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

With the mpi module and mpif.h, the Fortran callback functions are:

```
SUBROUTINE WIN_COPY_ATTR_FUNCTION(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_FUNCTION(WIN, WIN_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, 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) |

**C binding**

```
int MPI_Win_free_keyval(int *win_keyval)
```

**F08 binding**

```
MPI_Win_free_keyval(win_keyval, ierror)
    INTEGER, INTENT(INOUT) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR
```

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 |

**C binding**

```
int MPI_Win_set_attr(MPI_Win win, int win_keyval, void* attribute_val)
```

**F08 binding**

```
MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, INTENT(IN) ::  win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_WIN_GET_ATTR(win, win_keyval, attribute_val, flag)

| | | |
|---|---|---|
| IN | win | window to which the attribute is attached (handle) |
| IN | win_keyval | key value (integer) |
| OUT | attribute_val | attribute value, unless flag = false |
| OUT | flag | false if no attribute is associated with the key (logical) |

**C binding**

```
int MPI_Win_get_attr(MPI_Win win, int win_keyval, void* attribute_val,
             int *flag)
```

**F08 binding**

```
MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, INTENT(IN) ::  win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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_DELETE_ATTR(win, win_keyval)

| | | |
|---|---|---|
| INOUT | win | window from which the attribute is deleted (handle) |
| IN | win_keyval | key value (integer) |

**C binding**

```
int MPI_Win_delete_attr(MPI_Win win, int win_keyval)
```

**F08 binding**

```
MPI_Win_delete_attr(win, win_keyval, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
```

```
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
```

### 6.7.4 Datatypes

The new functions for caching on datatypes are:

MPI_TYPE_CREATE_KEYVAL(type_copy_attr_fn, type_delete_attr_fn, type_keyval, extra_state)

| | | |
|---|---|---|
| IN | type_copy_attr_fn | copy callback function for type_keyval (function) |
| IN | type_delete_attr_fn | delete callback function for type_keyval (function) |
| OUT | type_keyval | key value for future access (integer) |
| IN | extra_state | extra state for callback function |

**C binding**

```
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)
```

**F08 binding**

```
MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
    PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
    INTEGER, INTENT(OUT) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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
```

The argument type_copy_attr_fn may be specified as MPI_TYPE_NULL_COPY_FN or MPI_TYPE_DUP_FN from either 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 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 datatype,
             int type_keyval, void *attribute_val, void *extra_state);
```

With the mpi_f08 module, the Fortran callback functions are:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Datatype) ::  oldtype
      INTEGER ::  type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL ::  flag
```

and

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
  attribute_val, extra_state, ierror)
      TYPE(MPI_Datatype) ::  datatype
      INTEGER ::  type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
```

With the mpi module and mpif.h, the Fortran callback functions are:

```
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(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_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
             EXTRA_STATE, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, 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.

MPI_TYPE_FREE_KEYVAL(type_keyval)

| | | |
|---|---|---|
| INOUT | type_keyval | key value (integer) |

**C binding**

```
int MPI_Type_free_keyval(int *type_keyval)
```

**F08 binding**

```
MPI_Type_free_keyval(type_keyval, ierror)
    INTEGER, INTENT(INOUT) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_FREE_KEYVAL(TYPE_KEYVAL, IERROR)
    INTEGER TYPE_KEYVAL, IERROR
```

MPI_TYPE_SET_ATTR(datatype, type_keyval, attribute_val)

| | | |
|---|---|---|
| INOUT | datatype | datatype to which attribute will be attached (handle) |
| IN | type_keyval | key value (integer) |
| IN | attribute_val | attribute value |

**C binding**

```
int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
             void* attribute_val)
```

**F08 binding**

```
MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

MPI_TYPE_GET_ATTR(datatype, type_keyval, attribute_val, flag)

| | | |
|---|---|---|
| IN | datatype | 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) |

**C binding**

```
int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
             void* attribute_val, int *flag)
```

**F08 binding**

```
MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_GET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

MPI_TYPE_DELETE_ATTR(datatype, type_keyval)

| | | |
|---|---|---|
| INOUT | datatype | datatype from which the attribute is deleted (handle) |
| IN | type_keyval | key value (integer) |

**C binding**

```
int MPI_Type_delete_attr(MPI_Datatype datatype, int type_keyval)
```

**F08 binding**

```
MPI_Type_delete_attr(datatype, type_keyval, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_DELETE_ATTR(DATATYPE, TYPE_KEYVAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
```

### 6.7.5 Error Class for Invalid Keyval

Key values for attributes are system-allocated, by MPI_{TYPE,COMM,WIN}_CREATE_KEYVAL. Only such values can be passed to the functions that use key values as input arguments. In order to signal that an erroneous key value has been passed to one of these functions, there is a new MPI error class: MPI_ERR_KEYVAL. It can be returned by MPI_ATTR_PUT, MPI_ATTR_GET, MPI_ATTR_DELETE, MPI_KEYVAL_FREE, MPI_{TYPE,COMM,WIN}_DELETE_ATTR, MPI_{TYPE,COMM,WIN}_SET_ATTR, MPI_{TYPE,COMM,WIN}_GET_ATTR, MPI_{TYPE,COMM,WIN}_FREE_KEYVAL, MPI_COMM_DUP, MPI_COMM_IDUP, MPI_COMM_DISCONNECT, and MPI_COMM_FREE. The last four are included because keyval is an argument to the copy and delete functions for attributes.

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

```
/* key for this module's stuff: */
static int gop_key = MPI_KEYVAL_INVALID;

typedef struct
{
   int ref_count;          /* reference count */
   /* other stuff, whatever else we want */
} gop_stuff_type;

void Efficient_Collective_Op(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. */
    } else
        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 */

int gop_stuff_destructor(MPI_Comm comm, int keyval, void *gop_stuffP,
                         void *extra)
{
  gop_stuff_type *gop_stuff = (gop_stuff_type *)gop_stuffP;
  if (keyval != gop_key) { /* abort -- programming error */ }

  /* The group's being freed removes one reference to gop_stuff */
  gop_stuff->ref_count -= 1;

  /* If no references remain, then free the storage */
  if (gop_stuff->ref_count == 0) {
    free((void *)gop_stuff);
  }
  return MPI_SUCCESS;
}

/* The following routine is called by MPI when a group is copied */
int gop_stuff_copier(MPI_Comm comm, int keyval, void *extra,
               void *gop_stuff_inP, void *gop_stuff_outP, int *flag)
{
  gop_stuff_type *gop_stuff_in = (gop_stuff_type *)gop_stuff_inP;
  gop_stuff_type **gop_stuff_out = (gop_stuff_type **)gop_stuff_outP;
  if (keyval != gop_key) { /* abort -- programming error */ }

  /* The new group adds one reference to this gop_stuff */
  gop_stuff_in->ref_count += 1;
  *gop_stuff_out = gop_stuff_in;
  return MPI_SUCCESS;
}
```

## 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 binding**

```
int MPI_Comm_set_name(MPI_Comm comm, const char *comm_name)
```

**F08 binding**

```
MPI_Comm_set_name(comm, comm_name, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    CHARACTER(LEN=*), INTENT(IN) ::  comm_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) 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 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 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) |

**C binding**

```
int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)
```

**F08 binding**

```
MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_GET_NAME(COMM, COMM_NAME, RESULTLEN, IERROR)
    INTEGER COMM, RESULTLEN, IERROR
    CHARACTER*(*) COMM_NAME
```

MPI_COMM_GET_NAME returns the last name which has previously been associated with the given communicator. The name may be set and retrieved from any language. The same name will be returned independent of the language used. 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 additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI_MAX_OBJECT_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_OBJECT_NAME.

If the user has not associated a name with a communicator, or an error occurs, MPI_COMM_GET_NAME will return an empty string (all spaces in Fortran, "" in C). The three predefined communicators will have predefined names associated with them. Thus, the names of MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if not MPI_COMM_NULL) will have the default of MPI_COMM_WORLD, MPI_COMM_SELF, and MPI_COMM_PARENT. The fact that the system may have chosen to give a default name to a communicator does not prevent the user from setting a name on the same communicator; doing this removes the old name and assigns the new one.

*Rationale.* We provide separate functions for setting and getting the name of a communicator, rather than simply providing a predefined attribute key for the following reasons:

- It is not, in general, possible to store a string as an attribute from Fortran.
- It is not easy to set up the delete function for a string attribute unless it is known to have been allocated from the heap.

- 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. The constant MPI_MAX_OBJECT_NAME also applies to these names.

MPI_TYPE_SET_NAME(datatype, type_name)

| | | |
|---|---|---|
| INOUT | datatype | datatype whose identifier is to be set (handle) |
| IN | type_name | the character string which is remembered as the name (string) |

**C binding**

```
int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)
```

**F08 binding**

```
MPI_Type_set_name(datatype, type_name, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    CHARACTER(LEN=*), INTENT(IN) :: type_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
    INTEGER DATATYPE, IERROR
    CHARACTER*(*) TYPE_NAME
```

MPI_TYPE_GET_NAME(datatype, type_name, resultlen)

| | | |
|---|---|---|
| IN | datatype | datatype whose name is to be returned (handle) |
| OUT | type_name | the name previously stored on the datatype, or an empty string if no such name exists (string) |
| OUT | resultlen | length of returned name (integer) |

**C binding**

```
int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,
             int *resultlen)
```

**F08 binding**

```
MPI_Type_get_name(datatype, type_name, resultlen, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) ::  type_name
    INTEGER, INTENT(OUT) ::  resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER DATATYPE, RESULTLEN, IERROR
    CHARACTER*(*) TYPE_NAME
```

Named predefined datatypes have the default names of the datatype name. For example, MPI_WCHAR has the default name of MPI_WCHAR.

The following functions are used for setting and getting names of windows. The constant MPI_MAX_OBJECT_NAME also applies to these names.

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) |

**C binding**

```
int MPI_Win_set_name(MPI_Win win, const char *win_name)
```

**F08 binding**

```
MPI_Win_set_name(win, win_name, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    CHARACTER(LEN=*), INTENT(IN) ::  win_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) 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 an empty string if no such name exists (string) |
| OUT | resultlen | length of returned name (integer) |

**C binding**

```
int MPI_Win_get_name(MPI_Win win, char *win_name, int *resultlen)
```

**F08 binding**

```
MPI_Win_get_name(win, win_name, resultlen, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) ::  win_name
    INTEGER, INTENT(OUT) ::  resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
    CHARACTER*(*) WIN_NAME
```

## 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, communicator creation is 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 with machine-independent mapping and communication on virtual process topologies.

*Rationale.* Though physical mapping is not discussed, the existence of the virtual topology information may be used as advice by the runtime system. There are well-known techniques for mapping grid/torus structures to hardware topologies such as hypercubes or grids. For more complicated graph structures good heuristics often yield nearly optimal results [44]. 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 [11, 12].

Besides possible performance benefits, the virtual topology can function as a convenient, process-naming structure, with significant benefits for program readability and notational power in message-passing programming. (*End of rationale.*)

## 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.

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

MPI supports three topology types: **Cartesian**, **graph**, and **distributed graph**. The function MPI_CART_CREATE is used to create Cartesian topologies, the function MPI_GRAPH_CREATE is used to create graph topologies, and the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE are used to create distributed graph topologies. These topology creation functions are collective. As with other collective calls, the program must be written to work correctly, whether the call synchronizes or not.

The topology creation functions take as input an existing communicator comm_old, which defines the set of processes on which the topology is to be mapped. For MPI_GRAPH_CREATE and MPI_CART_CREATE, all input arguments must have identical values on all processes of the group of comm_old. When calling MPI_GRAPH_CREATE, each process specifies all nodes and edges in the graph. In contrast, the functions MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE are used to specify the graph in a distributed fashion, whereby each process only specifies a subset of the edges in the graph such that the entire graph structure is defined collectively across the set of processes. Therefore the processes provide different values for the arguments specifying the graph. However, all processes must give the same value for reorder and the info argument. In all cases, a new communicator comm_topol is created that carries the topological structure as cached information (see Chapter 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.

MPI defines functions to query a communicator for topology information. The function MPI_TOPO_TEST is used to query for the type of topology associated with a communicator. Depending on the topology type, different information can be extracted. For a graph topology, the functions MPI_GRAPHDIMS_GET and MPI_GRAPH_GET return the values that were specified in the call to MPI_GRAPH_CREATE. Additionally, the functions MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS can be used to obtain the neighbors of an arbitrary node in the graph. For a distributed graph topology, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS can be used to obtain the neighbors of the calling process. For a Cartesian topology, the functions MPI_CARTDIM_GET and MPI_CART_GET return the values that were specified in the call to MPI_CART_CREATE. Additionally, the functions MPI_CART_RANK and MPI_CART_COORDS translate Cartesian coordinates into a group rank, and vice-versa. The function MPI_CART_SHIFT provides the information needed to communicate with neighbors along a Cartesian dimension. All of these query functions are local.

For Cartesian topologies, the function MPI_CART_SUB can be used to extract a Cartesian subspace (analogous to MPI_COMM_SPLIT). This function is collective over the input communicator's group.

The two additional functions, MPI_GRAPH_MAP and MPI_CART_MAP, are, in general, not called by the user directly. However, together with the communicator manipulation functions presented in Chapter 6, they are sufficient to implement all other topology functions. Section 7.5.8 outlines such an implementation.

The neighborhood collective communication routines MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, and MPI_NEIGHBOR_ALLTOALLW communicate with the nearest neighbors on the topology associated with the communicator. The nonblocking variants are MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and

MPI_INEIGHBOR_ALLTOALLW.

## 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 | number of dimensions of Cartesian grid (integer) |
| IN | dims | integer array of size ndims specifying the number of processes in each dimension (array of positive integers) |
| IN | periods | logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension (array of logicals) |
| IN | reorder | ranking may be reordered (true) or not (false) (logical) |
| OUT | comm_cart | communicator with new Cartesian topology (handle) |

**C binding**

```
int MPI_Cart_create(MPI_Comm comm_old, const int ndims, const int dims[],
             int periods[], int reorder, MPI_Comm *comm_cart)
```

**F08 binding**

```
MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  ndims, dims(ndims)
    LOGICAL, INTENT(IN) ::  periods(ndims), reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_cart
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

MPI_CART_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_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_COMM_SPLIT. If ndims is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if ndims is negative.

### 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.

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 (array of positive integers) |

**C binding**

```
int MPI_Dims_create(int nnodes, int ndims, int dims[])
```

**F08 binding**

```
MPI_Dims_create(nnodes, ndims, dims, ierror)
    INTEGER, INTENT(IN) :: nnodes, ndims
    INTEGER, INTENT(INOUT) :: dims(ndims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

The entries in the array dims are set to describe a Cartesian grid with ndims dimensions and a total of nnodes nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array dims. If dims[i] is set to a positive number, the routine will not modify the number of nodes in dimension i; only those entries where dims[i] $= 0$ are modified by the call.

Negative input values of dims[i] are erroneous. An error will occur if nnodes is not a multiple of

$$\prod_{i,dims[i]\neq 0} dims[i].$$

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. If ndims is zero and nnodes is one, MPI_DIMS_CREATE returns MPI_SUCCESS.

**Example 7.1**

| dims before call | function call | dims on return |
|---|---|---|
| (0,0) | MPI_DIMS_CREATE(6, 2, dims) | (3,2) |
| (0,0) | MPI_DIMS_CREATE(7, 2, dims) | (7,1) |
| (0,3,0) | MPI_DIMS_CREATE(6, 3, dims) | (2,3,1) |
| (0,3,0) | MPI_DIMS_CREATE(7, 3, dims) | erroneous call |

### 7.5.3 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) (array of integers) |
| IN | edges | array of integers describing graph edges (see below) (array of non-negative integers) |
| IN | reorder | ranking may be reordered (true) or not (false) (logical) |
| OUT | comm_graph | communicator with graph topology added (handle) |

**C binding**

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],
             const int edges[], int reorder, MPI_Comm *comm_graph)
```

**F08 binding**

```
MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
             ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  nnodes, index(nnodes), edges(*)
    LOGICAL, INTENT(IN) ::  reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_graph
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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_CREATE returns a handle to a new communicator to which the graph 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. If the size, nnodes, of the graph is smaller than the size of the group of comm_old, then some processes are returned MPI_COMM_NULL, in analogy to MPI_CART_CREATE and MPI_COMM_SPLIT. If the graph is empty, i.e., nnodes == 0, then MPI_COMM_NULL is returned in all processes. The call is erroneous if it specifies a graph that is larger than the group size of the input communicator.

The three parameters nnodes, index and edges define the graph structure. nnodes is the number of nodes of the graph. The nodes are numbered from 0 to nnodes-1. The i-th entry of array index stores the total number of neighbors of the first i graph nodes. The lists of neighbors of nodes 0, 1, ..., nnodes-1 are stored in consecutive locations in array edges. The array edges is a flattened representation of the edge lists. The total number of entries in index is nnodes and the total number of entries in edges is equal to the number of graph edges.

The definitions of the arguments nnodes, index, and edges are illustrated with the following simple example.

**Example 7.2**

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix:

| process | neighbors |
|---|---|
| 0 | 1, 3 |
| 1 | 0 |
| 2 | 3 |
| 3 | 0, 2 |

Then, the input arguments are:

| | |
|---|---|
| nnodes = | 4 |
| index = | 2, 3, 4, 6 |
| edges = | 1, 3, 0, 3, 0, 2 |

Thus, in C, `index[0]` is the degree of node zero, and `index[i] - index[i-1]` is the degree of node `i, i=1, ..., nnodes-1`; the list of neighbors of node zero is stored in `edges[j]`, for $0 \leq$ `j` $\leq$ `index[0]` $-1$ and the list of neighbors of node `i`, `i` $>$ `0`, is stored in `edges[j]`, `index[i-1]` $\leq$ `j` $\leq$ `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(j)`, for `1` $\leq$ `j` $\leq$ `index(1)` and the list of neighbors of node `i`, `i` $>$ `0`, is stored in `edges(j)`, `index(i)+1` $\leq$ `j` $\leq$ `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 Distributed Graph Constructor

MPI_GRAPH_CREATE requires that each process passes the full (global) communication graph to the call. This limits the scalability of this constructor. With the distributed graph interface, the communication graph is specified in a fully distributed fashion. Each process specifies only the part of the communication graph of which it is aware. Typically, this could be the set of processes from which the process will eventually receive or get data, or the set of processes to which the process will send or put data, or some combination of such edges. Two different interfaces can be used to create a distributed graph topology. MPI_DIST_GRAPH_CREATE_ADJACENT creates a distributed graph communicator with each process specifying each of its incoming and outgoing (adjacent) edges in the logical communication graph and thus requires minimal communication during creation. MPI_DIST_GRAPH_CREATE provides full flexibility such that any process can indicate that communication will occur between any pair of processes in the graph.

To provide better possibilities for optimization by the MPI library, the distributed graph constructors permit weighted communication edges and take an info argument that can further influence process reordering or other optimizations performed by the MPI library. For example, hints can be provided on how edge weights are to be interpreted, the quality of the reordering, and/or the time permitted for the MPI library to process the graph.

MPI_DIST_GRAPH_CREATE_ADJACENT(comm_old, indegree, sources, sourceweights, outdegree, destinations, destweights, info, reorder, comm_dist_graph)

| | | |
|---|---|---|
| IN | comm_old | input communicator (handle) |
| IN | indegree | size of sources and sourceweights arrays (non-negative integer) |
| IN | sources | ranks of processes for which the calling process is a destination (array of non-negative integers) |
| IN | sourceweights | weights of the edges into the calling process (array of non-negative integers) |
| IN | outdegree | size of destinations and destweights arrays (non-negative integer) |
| IN | destinations | ranks of processes for which the calling process is a source (array of non-negative integers) |
| IN | destweights | weights of the edges out of the calling process (array of non-negative integers) |
| IN | info | hints on optimization and interpretation of weights (handle) |
| IN | reorder | the ranks may be reordered (true) or not (false) (logical) |
| OUT | comm_dist_graph | communicator with distributed graph topology (handle) |

**C binding**

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
             const int sources[], const int sourceweights[], int outdegree,
             const int destinations[], const int destweights[],
             MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

**F08 binding**

```
MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
             outdegree, destinations, destweights, info, reorder,
             comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: indegree, sources(indegree), sourceweights(*),
    outdegree, destinations(outdegree), destweights(*)
    TYPE(MPI_Info), INTENT(IN) :: info
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
             OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
             COMM_DIST_GRAPH, IERROR)
```

```
INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
LOGICAL REORDER
```

MPI_DIST_GRAPH_CREATE_ADJACENT returns a handle to a new communicator to which the distributed graph topology information is attached. Each process passes all information about its incoming and outgoing edges in the virtual distributed graph topology. The calling processes must ensure that each edge of the graph is described in the source and in the destination process with the same weights. If there are multiple edges for a given (source,dest) pair, then the sequence of the weights of these edges does not matter. The complete communication topology is the combination of all edges shown in the sources arrays of all processes in comm_old, which must be identical to the combination of all edges shown in the destinations arrays. Source and destination ranks must be process ranks of comm_old. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that have specified indegree and outdegree as zero and thus do not occur as source or destination rank in the graph specification) are allowed.

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE_ADJACENT is collective.

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not all processes of comm_old. If the graph is weighted but indegree or outdegree is zero, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to sourceweights or destweights respectively. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

*Advice to users.* In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI_UNWEIGHTED to the implementation. In this case MPI_WEIGHTS_EMPTY should be used instead. (*End of advice to users.*)

*Advice to implementors.* It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (*End of advice to implementors.*)

*Rationale.* To ensure backward compatibility, MPI_UNWEIGHTED may still be implemented as NULL. See Annex B.3. (*End of rationale.*)

The meaning of the info and reorder arguments is defined in the description of the following routine.

MPI_DIST_GRAPH_CREATE(comm_old, n, sources, degrees, destinations, weights, info, reorder, comm_dist_graph)

| | | |
|---|---|---|
| IN | comm_old | input communicator (handle) |
| IN | n | number of source nodes for which this process specifies edges (integer) |
| IN | sources | array containing the n source nodes forwhich this process specifies edges (array of non-negative integers) |
| IN | degrees | array specifying the number of destinations for each source node in the source node array (array of non-negative integers) |
| IN | destinations | destination nodes for the source nodes in the source node array (array of non-negative integers) |
| IN | weights | weights for source to destination edges (array of non-negative integers) |
| IN | info | hints on optimization and interpretation of weights (handle) |
| IN | reorder | the ranks may be reordered (true) or not (false) (logical) |
| OUT | comm_dist_graph | communicator with distributed graph topology added (handle) |

**C binding**

```
int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],
             const int degrees[], const int destinations[],
             const int weights[], MPI_Info info, int reorder,
             MPI_Comm *comm_dist_graph)
```

**F08 binding**

```
MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
             info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: n, sources(n), degrees(n), destinations(*),
    weights(*)
    TYPE(MPI_Info), INTENT(IN) :: info
    LOGICAL, INTENT(IN) :: reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
             INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER
```

MPI_DIST_GRAPH_CREATE returns a handle to a new communicator to which the distributed graph topology information is attached. Concretely, each process calls the constructor with a set of directed (source,destination) communication edges as described below. Every process passes an array of n source nodes in the sources array. For each source node, a non-negative number of destination nodes is specified in the degrees array. The destination nodes are stored in the corresponding consecutive segment of the destinations array. More precisely, if the i-th node in sources is s, this specifies degrees[i] edges (s,d) with d of the j-th such edge stored in destinations[degrees[0]+...+degrees[i-1]+j]. The weight of this edge is stored in weights[degrees[0]+...+degrees[i-1]+j]. Both the sources and the destinations arrays may contain the same node more than once, and the order in which nodes are listed as destinations or sources is not significant. Similarly, different processes may specify edges with the same source and destination nodes. Source and destination nodes must be process ranks of comm_old. Different processes may specify different numbers of source and destination nodes, as well as different source to destination edges. This allows a fully distributed specification of the communication graph. Isolated processes (i.e., processes with no outgoing or incoming edges, that is, processes that do not occur as source or destination node in the graph specification) are allowed.

The call creates a new communicator comm_dist_graph of distributed graph topology type to which topology information has been attached. The number of processes in comm_dist_graph is identical to the number of processes in comm_old. The call to MPI_DIST_GRAPH_CREATE is collective.

If reorder = false, all processes will have the same rank in comm_dist_graph as in comm_old. If reorder = true then the MPI library is free to remap to other processes (of comm_old) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering.

Weights are specified as non-negative integers and can be used to influence the process remapping strategy and other internal MPI optimizations. For instance, approximate count arguments of later communication calls along specific edges could be used as their edge weights. Multiplicity of edges can likewise indicate more intense communication between pairs of processes. However, the exact meaning of edge weights is not specified by the MPI standard and is left to the implementation. In C or Fortran, an application can supply the special value MPI_UNWEIGHTED for the weight array to indicate that all edges have the same (effectively no) weight. It is erroneous to supply MPI_UNWEIGHTED for some but not all processes of comm_old. If the graph is weighted but n = 0, then MPI_WEIGHTS_EMPTY or any arbitrary array may be passed to weights. Note that MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are not special weight values; rather they are special values for the total array argument. In Fortran, MPI_UNWEIGHTED and MPI_WEIGHTS_EMPTY are objects like MPI_BOTTOM (not usable for initialization or assignment). See Section 2.5.4.

*Advice to users.* In the case of an empty weights array argument passed while constructing a weighted graph, one should not pass NULL because the value of MPI_UNWEIGHTED may be equal to NULL. The value of this argument would then be indistinguishable from MPI_UNWEIGHTED to the implementation. MPI_WEIGHTS_EMPTY should be used instead. (*End of advice to users.*)

*Advice to implementors.* It is recommended that MPI_UNWEIGHTED not be implemented as NULL. (*End of advice to implementors.*)

*Rationale.* To ensure backward compatibility, MPI_UNWEIGHTED may still be implemented as NULL. See Annex B.3. (*End of rationale.*)

The meaning of the weights argument can be influenced by the info argument. Info arguments can be used to guide the mapping; possible options include minimizing the maximum number of edges between processes on different SMP nodes, or minimizing the sum of all such edges. An MPI implementation is not obliged to follow specific hints, and it is valid for an MPI implementation not to do any reordering. An MPI implementation may specify more info key-value pairs. All processes must specify the same set of key-value info pairs.

*Advice to implementors.* MPI implementations must document any additionally supported key-value info pairs. MPI_INFO_NULL is always valid, and may indicate the default creation of the distributed graph topology to the MPI library.

An implementation does not explicitly need to construct the topology from its distributed parts. However, all processes can construct the full topology from the distributed specification and use this in a call to MPI_GRAPH_CREATE to create the topology. This may serve as a reference implementation of the functionality, and may be acceptable for small communicators. However, a scalable high-quality implementation would save the topology graph in a distributed way. (*End of advice to implementors.*)

**Example 7.3** As for Example 7.2, assume there are four processes 0, 1, 2, 3 with the following adjacency matrix and unit edge weights:

| process | neighbors |
|---|---|
| 0 | 1, 3 |
| 1 | 0 |
| 2 | 3 |
| 3 | 0, 2 |

With MPI_DIST_GRAPH_CREATE, this graph could be constructed in many different ways. One way would be that each process specifies its outgoing edges. The arguments per process would be:

| process | n | sources | degrees | destinations | weights |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 2 | 1,3 | 1,1 |
| 1 | 1 | 1 | 1 | 0 | 1 |
| 2 | 1 | 2 | 1 | 3 | 1 |
| 3 | 1 | 3 | 2 | 0,2 | 1,1 |

Another way would be to pass the whole graph on process 0, which could be done with the following arguments per process:

| process | n | sources | degrees | destinations | weights |
|---|---|---|---|---|---|
| 0 | 4 | 0,1,2,3 | 2,1,1,2 | 1,3,0,3,0,2 | 1,1,1,1,1,1 |
| 1 | 0 | - | - | - | - |
| 2 | 0 | - | - | - | - |
| 3 | 0 | - | - | - | |

In both cases above, the application could supply MPI_UNWEIGHTED instead of explicitly providing identical weights.

MPI_DIST_GRAPH_CREATE_ADJACENT could be used to specify this graph using the following arguments:

| process | indegree | sources | sourceweights | outdegree | destinations | destweights |
|---|---|---|---|---|---|---|
| 0 | 2 | 1,3 | 1,1 | 2 | 1,3 | 1,1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 2 | 1 | 3 | 1 | 1 | 3 | 1 |
| 3 | 2 | 0,2 | 1,1 | 2 | 0,2 | 1,1 |

**Example 7.4** A two-dimensional PxQ torus where all processes communicate along the dimensions and along the diagonal edges. This cannot be modeled with Cartesian topologies, but can easily be captured with MPI_DIST_GRAPH_CREATE as shown in the following code. In this example, the communication along the dimensions is twice as heavy as the communication along the diagonals:

```
/*
Input:     dimensions P, Q
Condition: number of processes equal to P*Q; otherwise only
           ranks smaller than P*Q participate
*/
int rank, x, y;
int sources[1], degrees[1];
int destinations[8], weights[8];
MPI_Comm comm_dist_graph;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

/* get x and y dimension */
y=rank/P; x=rank%P;

/* get my communication partners along x dimension */
destinations[0] = P*y+(x+1)%P; weights[0] = 2;
destinations[1] = P*y+(P+x-1)%P; weights[1] = 2;

/* get my communication partners along y dimension */
destinations[2] = P*((y+1)%Q)+x; weights[2] = 2;
destinations[3] = P*((Q+y-1)%Q)+x; weights[3] = 2;

/* get my communication partners along diagonals */
destinations[4] = P*((y+1)%Q)+(x+1)%P; weights[4] = 1;
destinations[5] = P*((Q+y-1)%Q)+(x+1)%P; weights[5] = 1;
destinations[6] = P*((y+1)%Q)+(P+x-1)%P; weights[6] = 1;
destinations[7] = P*((Q+y-1)%Q)+(P+x-1)%P; weights[7] = 1;

sources[0] = rank;
degrees[0] = 8;
```

```
MPI_Dist_graph_create(MPI_COMM_WORLD, 1, sources, degrees, destinations,
                      weights, MPI_INFO_NULL, 1, &comm_dist_graph);
```

### 7.5.5 Topology Inquiry Functions

If a topology has been defined with one of the above functions, then the topology information can be looked up using inquiry functions. They all are local calls.

MPI_TOPO_TEST(comm, status)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | status | topology type of communicator comm (integer) |

**C binding**

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

**F08 binding**

```
MPI_Topo_test(comm, status, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER 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_DIST_GRAPH | distributed graph 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 (same as number of processes in the group) (integer) |
| OUT | nedges | number of edges in graph (integer) |

**C binding**

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

**F08 binding**

```
MPI_Graphdims_get(comm, nnodes, nedges, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    INTEGER, INTENT(OUT) ::  nnodes, nedges
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
```

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.

MPI_GRAPH_GET(comm, maxindex, maxedges, index, 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) (array of integers) |
| OUT | edges | array of integers containing the graph structure (array of non-negative integers) |

**C binding**

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],
              int edges[])
```

**F08 binding**

```
MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  maxindex, maxedges
    INTEGER, INTENT(OUT) ::  index(maxindex), edges(maxedges)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

MPI_CARTDIM_GET(comm, ndims)

| | | |
|---|---|---|
| IN | comm | communicator with Cartesian structure (handle) |
| OUT | ndims | number of dimensions of the Cartesian structure (integer) |

**C binding**

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

**F08 binding**

```
MPI_Cartdim_get(comm, ndims, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: ndims
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

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 positive integers) |
| OUT | periods | periodicity (true/false) for each Cartesian dimension (array of logicals) |
| OUT | coords | coordinates of calling process in Cartesian structure (array of integers) |

**C binding**

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
             int coords[])
```

**F08 binding**

```
MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: maxdims
    INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
    LOGICAL, INTENT(OUT) :: periods(maxdims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
    LOGICAL PERIODS(*)
```

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 (array of integers) |
| OUT | rank | rank of specified process (non-negative integer) |

**C binding**

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

**F08 binding**

```
MPI_Cart_rank(comm, coords, rank, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  coords(*)
    INTEGER, INTENT(OUT) ::  rank
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR
```

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, coords is not significant and 0 is returned in rank.

MPI_CART_COORDS(comm, rank, maxdims, coords)

| | | |
|---|---|---|
| IN | comm | communicator with Cartesian structure (handle) |
| IN | rank | rank of a process within group of comm (non-negative integer) |
| IN | maxdims | length of vector coords in the calling program (integer) |
| OUT | coords | integer array (of size maxdims) containing the Cartesian coordinates of specified process (array of integers) |

**C binding**

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

**F08 binding**

```
MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  rank, maxdims
    INTEGER, INTENT(OUT) ::  coords(maxdims)
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

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(comm, rank, nneighbors)

| | | |
|---|---|---|
| IN | comm | communicator with graph topology (handle) |
| IN | rank | rank of process in group of comm (non-negative integer) |
| OUT | nneighbors | number of neighbors of specified process (integer) |

**C binding**

```
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

**F08 binding**

```
MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: rank
    INTEGER, INTENT(OUT) :: nneighbors
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

| | | |
|---|---|---|
| IN | comm | communicator with graph topology (handle) |
| IN | rank | rank of process in group of comm (non-negative integer) |
| IN | maxneighbors | size of array neighbors (integer) |
| OUT | neighbors | ranks of processes that are neighbors to specified process (array of non-negative integers) |

**C binding**

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
              int neighbors[])
```

**F08 binding**

```
MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
```

```
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  rank, maxneighbors
    INTEGER, INTENT(OUT) ::  neighbors(maxneighbors)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS provide adjacency information for a graph topology. The returned count and array of neighbors for the queried rank will both include *all* neighbors and reflect the same edge ordering as was specified by the original call to MPI_GRAPH_CREATE. Specifically, MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS will return values based on the original index and edges array passed to MPI_GRAPH_CREATE (for the purpose of this example, we assume that index[-1] is zero):

- The number of neighbors (nneighbors) returned from MPI_GRAPH_NEIGHBORS_COUNT will be (index[rank] - index[rank-1]).
- The neighbors array returned from MPI_GRAPH_NEIGHBORS will be edges[index[rank-1]] through edges[index[rank]-1].

**Example 7.5**

Assume there are four processes 0, 1, 2, 3 with the following adjacency matrix (note that some neighbors are listed multiple times):

| process | neighbors |
|---|---|
| 0 | 1, 1, 3 |
| 1 | 0, 0 |
| 2 | 3 |
| 3 | 0, 2, 2 |

Thus, the input arguments to MPI_GRAPH_CREATE are:

| | |
|---|---|
| nnodes = | 4 |
| index = | 3, 5, 6, 9 |
| edges = | 1, 1, 3, 0, 0, 3, 0, 2, 2 |

Therefore, calling MPI_GRAPH_NEIGHBORS_COUNT and MPI_GRAPH_NEIGHBORS for each of the 4 processes will return:

| Input rank | Count | Neighbors |
|---|---|---|
| 0 | 3 | 1, 1, 3 |
| 1 | 2 | 0, 0 |
| 2 | 1 | 3 |
| 3 | 3 | 0, 2, 2 |

**Example 7.6**

Suppose that comm is a communicator with a shuffle-exchange topology. The group has $2^n$ members. Each process is labeled by $a_1, \ldots, a_n$ with $a_i \in \{0, 1\}$, and has three neighbors: exchange$(a_1, \ldots, a_n) = a_1, \ldots, a_{n-1}, \bar{a}_n$ $(\bar{a} = 1 - a)$, shuffle$(a_1, \ldots, a_n) = a_2, \ldots, a_n, a_1$, and unshuffle$(a_1, \ldots, a_n) = a_n, a_1, \ldots, a_{n-1}$. The graph adjacency list is illustrated below for $n = 3$.

| node | | exchange neighbors(1) | shuffle neighbors(2) | unshuffle neighbors(3) |
|---|---|---|---|---|
| 0 | (000) | 1 | 0 | 0 |
| 1 | (001) | 0 | 2 | 4 |
| 2 | (010) | 3 | 4 | 1 |
| 3 | (011) | 2 | 6 | 5 |
| 4 | (100) | 5 | 1 | 2 |
| 5 | (101) | 4 | 3 | 6 |
| 6 | (110) | 7 | 5 | 3 |
| 7 | (111) | 6 | 7 | 7 |

Suppose that the communicator comm has this topology associated with it. The following code fragment cycles through the three types of neighbors and performs an appropriate permutation for each.

```
!  assume: each process has stored a real number A.
!  extract neighborhood information
      CALL MPI_COMM_RANK(comm, myrank, ierr)
      CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
!  perform exchange permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(1), 0, &
           neighbors(1), 0, comm, status, ierr)
!  perform shuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(2), 0, &
           neighbors(3), 0, comm, status, ierr)
!  perform unshuffle permutation
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, neighbors(3), 0, &
           neighbors(2), 0, comm, status, ierr)
```

MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS provide adjacency information for a distributed graph topology.

MPI_DIST_GRAPH_NEIGHBORS_COUNT(comm, indegree, outdegree, weighted)

| | | |
|---|---|---|
| IN | comm | communicator with distributed graph topology (handle) |
| OUT | indegree | number of edges into this process (non-negative integer) |
| OUT | outdegree | number of edges out of this process (non-negative integer) |
| OUT | weighted | false if MPI_UNWEIGHTED was supplied during creation, true otherwise (logical) |

**C binding**

```
int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
              int *outdegree, int *weighted)
```

**F08 binding**

```
MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  indegree, outdegree
    LOGICAL, INTENT(OUT) ::  weighted
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
    INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
    LOGICAL WEIGHTED
```

MPI_DIST_GRAPH_NEIGHBORS(comm, maxindegree, sources, sourceweights, maxoutdegree, destinations, destweights)

| | | |
|---|---|---|
| IN | comm | communicator with distributed graph topology (handle) |
| IN | maxindegree | size of sources and sourceweights arrays (integer) |
| OUT | sources | processes for which the calling process is a destination (array of non-negative integers) |
| OUT | sourceweights | weights of the edges into the calling process (array of non-negative integers) |
| IN | maxoutdegree | size of destinations and destweights arrays (integer) |
| OUT | destinations | processes for which the calling process is a source (array of non-negative integers) |
| OUT | destweights | weights of the edges out of the calling process (array of non-negative integers) |

**C binding**

```
int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
             int sourceweights[], int maxoutdegree, int destinations[],
             int destweights[])
```

**F08 binding**

```
MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
             maxoutdegree, destinations, destweights, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  maxindegree, maxoutdegree
    INTEGER, INTENT(OUT) ::  sources(maxindegree),
    destinations(maxoutdegree)
    INTEGER ::  sourceweights(*), destweights(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
             MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
    INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
    DESTINATIONS(*), DESTWEIGHTS(*), IERROR
```

These calls are local. The number of edges into and out of the process returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT are the total number of such edges given in the call to MPI_DIST_GRAPH_CREATE_ADJACENT or MPI_DIST_GRAPH_CREATE (potentially by processes other than the calling process in the case of MPI_DIST_GRAPH_CREATE). Multiply defined edges are all counted and returned by MPI_DIST_GRAPH_NEIGHBORS in some order. If MPI_UNWEIGHTED is supplied for sourceweights or destweights or both, or if MPI_UNWEIGHTED was supplied during the construction of the graph then no weight information is returned in that array or those arrays. If the communicator was created with MPI_DIST_GRAPH_CREATE_ADJACENT then for each rank in comm, the order of the values in sources and destinations is identical to the input that was used by the process with the same rank in comm_old in the creation call. If the communicator was created with MPI_DIST_GRAPH_CREATE then the only requirement on the order of values in sources and destinations is that two calls to the routine with same input argument comm will return the same sequence of edges. If maxindegree or maxoutdegree is smaller than the numbers returned by MPI_DIST_GRAPH_NEIGHBORS_COUNT, then only the first part of the full list is returned.

> *Advice to implementors.* Since the query calls are defined to be local, each process needs to store the list of its neighbors with incoming and outgoing edges. Communication is required at the collective MPI_DIST_GRAPH_CREATE call in order to compute the neighbor lists for each process from the distributed graph specification. (*End of advice to implementors.*)

### 7.5.6 Cartesian Shift Coordinates

If the process topology is a Cartesian structure, an MPI_SENDRECV operation may 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(comm, direction, disp, rank_source, rank_dest)

| | | |
|---|---|---|
| IN | comm | communicator with Cartesian structure (handle) |
| IN | direction | coordinate dimension of shift (integer) |
| IN | disp | displacement (> 0: upwards shift, < 0: downwards shift) (positive integer) |
| OUT | rank_source | rank of source process (non-negative integer) |
| OUT | rank_dest | rank of destination process (non-negative integer) |

**C binding**

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
              int *rank_source, int *rank_dest)
```

**F08 binding**

```
MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: direction, disp
    INTEGER, INTENT(OUT) :: rank_source, rank_dest
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR)
    INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR
```

The direction argument indicates the coordinate dimension to be traversed by the shift. The dimensions are numbered from 0 to ndims-1, where ndims is the number of dimensions.

Depending on the periodicity of the Cartesian group in the specified coordinate direction, MPI_CART_SHIFT provides the identifiers for a circular or an end-off shift. In the case of an end-off shift, the value MPI_PROC_NULL may be returned in rank_source or rank_dest, indicating that the source or the destination for the shift is out of range.

It is erroneous to call MPI_CART_SHIFT with a direction that is either negative or greater than or equal to the number of dimensions in the Cartesian communicator. This implies that it is erroneous to call MPI_CART_SHIFT with a comm that is associated with a zero-dimensional Cartesian topology.

**Example 7.7**

The communicator, comm, has a two-dimensional, periodic, Cartesian topology associated with it. A two-dimensional array of REALs is stored one element per process, in variable A. One wishes to skew this array, by shifting column i (vertically, i.e., along the column) by i steps.

```
...
! find process rank
      CALL MPI_COMM_RANK(comm, rank, ierr)
! find Cartesian coordinates
      CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
! compute shift source and destination
      CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
! skew array
      CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0, source, 0, comm, &
                                status, ierr)
```

*Advice to users.* In Fortran, the dimension indicated by DIRECTION = i has DIMS(i+1) nodes, where DIMS is the array that was used to create the grid. In C, the dimension indicated by direction = i is the dimension specified by dims[i]. (*End of advice to users.*)

### 7.5.7 Partitioning of Cartesian Structures

MPI_CART_SUB(comm, remain_dims, newcomm)

| | | |
|---|---|---|
| IN | comm | communicator with Cartesian structure (handle) |
| IN | remain_dims | the i-th entry of remain_dims specifies whether the i-th dimension is kept in the subgrid (true) or is dropped (false) (array of logicals) |
| OUT | newcomm | communicator containing the subgrid that includes the calling process (handle) |

**C binding**

```
int MPI_Cart_sub(MPI_Comm comm, const int remain_dims[], MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(IN) :: remain_dims(*)
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
    LOGICAL REMAIN_DIMS(*)
```

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 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. (This function is closely related to MPI_COMM_SPLIT.)

**Example 7.8**

Assume that MPI_CART_CREATE(..., comm) has defined a $(2 \times 3 \times 4)$ grid. Let remain_dims = (true, false, true). Then a call to

```
MPI_CART_SUB(comm, remain_dims, comm_new);
```

will create three communicators each with eight processes in a $2 \times 4$ Cartesian topology. If remain_dims = (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.

### 7.5.8 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. The two calls are both local.

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 (array of positive integers) |
| IN | periods | logical array of size ndims specifying the periodicity specification in each coordinate direction (array of logicals) |
| OUT | newrank | reordered rank of the calling process; MPI_UNDEFINED if calling process does not belong to grid (non-negative integer) |

**C binding**

```
int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],
             const int periods[], int *newrank)
```

**F08 binding**

```
MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  ndims, dims(ndims)
    LOGICAL, INTENT(IN) ::  periods(ndims)
    INTEGER, INTENT(OUT) ::  newrank
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_CART_MAP(COMM, NDIMS, DIMS, PERIODS, NEWRANK, IERROR)
    INTEGER COMM, NDIMS, DIMS(*), NEWRANK, IERROR
    LOGICAL PERIODS(*)
```

MPI_CART_MAP computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering.

*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 $\neq$ MPI_UNDEFINED, color = MPI_UNDEFINED otherwise, and key = newrank. If ndims is zero then a zero-dimensional Cartesian topology is created.

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 function for graph structures is as follows.

MPI_GRAPH_MAP(comm, nnodes, index, edges, newrank)

| | | |
|---|---|---|
| IN | comm | input communicator (handle) |
| IN | nnodes | number of graph nodes (integer) |
| IN | index | integer array specifying the graph structure, see lushline MPI_GRAPH_CREATE (array of integers) |
| IN | edges | integer array specifying the graph structure (array of non-negative integers) |
| OUT | newrank | reordered rank of the calling process; MPI_UNDEFINED if the calling process does not belong to graph (non-negative integer) |

**C binding**

```
int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],
             const int edges[], int *newrank)
```

**F08 binding**

```
MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: nnodes, index(nnodes), edges(*)
    INTEGER, INTENT(OUT) :: newrank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
```

*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.*)

## 7.6 Neighborhood Collective Communication on Process Topologies

MPI process topologies specify a communication graph, but they implement no communication function themselves. Many applications require sparse nearest neighbor communications that can be expressed as graph topologies. We now describe several collective operations that perform communication along the edges of a process topology. All of these functions are collective; i.e., they must be called by all processes in the specified communicator. See Section 5 for an overview of other dense (global) collective communication operations and the semantics of collective operations.

If the graph was created with MPI_DIST_GRAPH_CREATE_ADJACENT with sources and destinations containing 0, ..., n-1, where n is the number of processes in the group of comm_old (i.e., the graph is fully connected and also includes an edge from each node to itself), then the sparse neighborhood communication routine performs the same data exchange as the corresponding dense (fully-connected) collective operation. In the case of a Cartesian communicator, only nearest neighbor communication is provided, corresponding to rank_source and rank_dest in MPI_CART_SHIFT with input disp=1.

*Rationale.* Neighborhood collective communications enable communication on a process topology. This high-level specification of data exchange among neighboring processes enables optimizations in the MPI library because the communication pattern is known statically (the topology). Thus, the implementation can compute optimized message schedules during creation of the topology [35]. This functionality can significantly simplify the implementation of neighbor exchanges [31]. (*End of rationale.*)

For a distributed graph topology, created with MPI_DIST_GRAPH_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined as the sequence returned by MPI_DIST_GRAPH_NEIGHBORS for destinations and sources, respectively. For a general graph topology, created with MPI_GRAPH_CREATE, the use of neighborhood collective communication is restricted to adjacency matrices, where the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix). In this case, the order of neighbors in the send and receive buffers is defined as the sequence of neighbors as returned by MPI_GRAPH_NEIGHBORS. Note that general graph topologies should generally be replaced by the distributed graph topologies.

For a Cartesian topology, created with MPI_CART_CREATE, the sequence of neighbors in the send and receive buffers at each process is defined by order of the dimensions, first the neighbor in the negative direction and then in the positive direction with displacement 1. The numbers of sources and destinations in the communication routines are 2*ndims with ndims defined in MPI_CART_CREATE. If a neighbor does not exist, i.e., at the border of a Cartesian topology in the case of a non-periodic virtual grid dimension (i.e., periods[...]==false), then this neighbor is defined to be MPI_PROC_NULL.

If a neighbor in any of the functions is MPI_PROC_NULL, then the neighborhood collective communication behaves like a point-to-point communication with MPI_PROC_NULL in this direction. That is, the buffer is still part of the sequence of neighbors but it is neither communicated nor updated.

### 7.6.1 Neighborhood Gather

In this function, each process $i$ gathers data items from each process $j$ if an edge $(j, i)$ exists in the topology graph, and each process $i$ sends the same data items to all processes $j$ where an edge $(i, j)$ exists. The send buffer is sent to each neighboring process and the $l$-th block in the receive buffer is received from the $l$-th neighbor.

MPI_NEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | number of elements received from each neighbor (non-negative integer) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |

**C binding**

```
int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
             MPI_Datatype sendtype, void *recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                         outdegree, dsts, MPI_UNWEIGHTED);
int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf, sendcount, sendtype,dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
            srcs[l],...);

MPI_Waitall(...);
```

Figure 7.1 shows the neighborhood gather communication of one process with outgoing neighbors $d_0 \ldots d_3$ and incoming neighbors $s_0 \ldots s_5$. The process will send its sendbuf to all four destinations (outgoing neighbors) and it will receive the contribution from all six sources (incoming neighbors) into separate locations of its receive buffer.

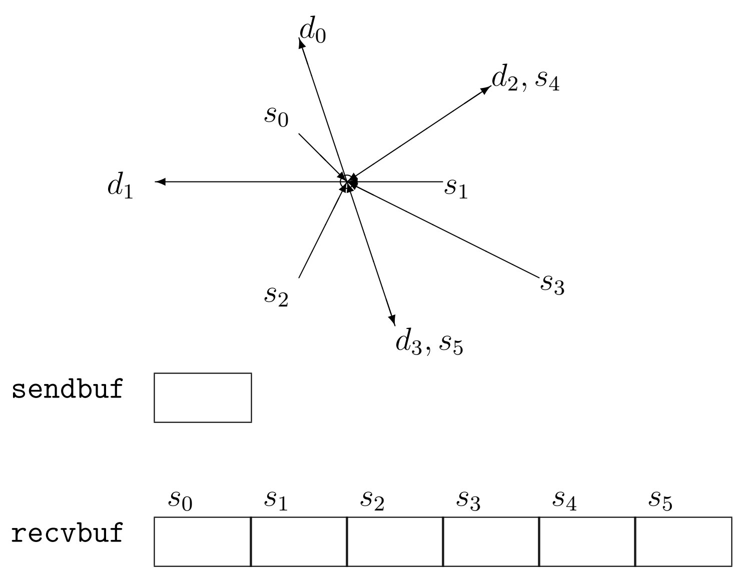


Figure 7.1: Neighborhood gather communication example.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at all other processes. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

*Rationale.* For optimization reasons, the same type signature is required independently of whether the topology graph is connected or not. (*End of rationale.*)

The "in place" option is not meaningful for this operation.

The vector variant of MPI_NEIGHBOR_ALLGATHER allows one to gather different numbers of elements from each neighbor.

MPI_NEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor (array of non-negative integers) |
| IN | displs | integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |

**C binding**

```
int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
             MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
             displs, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                         outdegree, dsts, MPI_UNWEIGHTED);
int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf, sendcount, sendtype, dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+displs[l]*extent(recvtype), recvcounts[l], recvtype,
            srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcount, sendtype, at process j must be equal to the type signature associated with recvcounts[l], recvtype at any other process with srcs[l]==j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data received from the l-th neighbor is placed into recvbuf beginning at offset displs[l] elements (in terms of the recvtype).

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

### 7.6.2 Neighbor Alltoall

In this function, each process $i$ receives data items from each process $j$ if an edge $(j, i)$ exists in the topology graph or Cartesian topology. Similarly, each process $i$ sends data items to all processes $j$ where an edge $(i, j)$ exists. This call is more general than MPI_NEIGHBOR_ALLGATHER in that different data items can be sent to each neighbor. The $k$-th block in send buffer is sent to the $k$-th neighboring process and the $l$-th block in the receive buffer is received from the $l$-th neighbor.

MPI_NEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | number of elements received from each neighbor (non-negative integer) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |

**C binding**

```
int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
             MPI_Datatype sendtype, void *recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                         outdegree, dsts, MPI_UNWEIGHTED);
int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
```

```
  MPI_Isend(sendbuf+k*sendcount*extent(sendtype), sendcount, sendtype,
            dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+l*recvcount*extent(recvtype), recvcount, recvtype,
            srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

The vector variant of MPI_NEIGHBOR_ALLTOALL allows sending/receiving different numbers of elements to and from each neighbor.

MPI_NEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor (array of non-negative integers) |
| IN | sdispls | integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which to send the outgoing data to neighbor j (array of integers) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor (array of non-negative integers) |
| IN | rdispls | integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |

**C binding**

```
int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
```

```
            const int recvcounts[], const int rdispls[],
            MPI_Datatype recvtype, MPI_Comm comm)
```

**F08 binding**

```
MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
            recvcounts, rdispls, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
    rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(*), DIMENSION(..) :: recvbuf
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
            RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                         outdegree, dsts, MPI_UNWEIGHTED);
int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf+sdispls[k]*extent(sendtype), sendcounts[k], sendtype,
          dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+rdispls[l]*extent(recvtype), recvcounts[l], recvtype,
          srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtype with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtype with srcs[l]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed. The data in the sendbuf beginning at offset sdispls[k] elements (in terms of the sendtype) is sent to the k-th outgoing neighbor. The data received from the l-th incoming neighbor is placed into recvbuf beginning at offset rdispls[l] elements (in terms of the recvtype).

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

MPI_NEIGHBOR_ALLTOALLW allows one to send and receive with different datatypes to and from each neighbor.

MPI_NEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor |
| IN | sdispls | integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers) |
| IN | sendtypes | array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor |
| IN | rdispls | integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers) |
| IN | recvtypes | array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles) |
| IN | comm | communicator with topology structure (handle) |

**C binding**

```
int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[],
             const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
             void* recvbuf, const int recvcounts[],
             const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
             MPI_Comm comm)
```

**F08 binding**

```
MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
```

```
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  sdispls(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtypes(*), recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
              IERROR
```

This function supports Cartesian communicators, graph communicators, and distributed graph communicators as described in Section 7.6. If comm is a distributed graph communicator, the outcome is as if each process executed sends to each of its outgoing neighbors and receives from each of its incoming neighbors:

```
MPI_Dist_graph_neighbors_count(comm, &indegree, &outdegree, &weighted);
int *srcs=(int*)malloc(indegree*sizeof(int));
int *dsts=(int*)malloc(outdegree*sizeof(int));
MPI_Dist_graph_neighbors(comm, indegree, srcs, MPI_UNWEIGHTED,
                         outdegree, dsts, MPI_UNWEIGHTED);
int k,l;

/* assume sendbuf and recvbuf are of type (char*) */
for(k=0; k<outdegree; ++k)
  MPI_Isend(sendbuf+sdispls[k], sendcounts[k], sendtypes[k], dsts[k],...);

for(l=0; l<indegree; ++l)
  MPI_Irecv(recvbuf+rdispls[l], recvcounts[l], recvtypes[l], srcs[l],...);

MPI_Waitall(...);
```

The type signature associated with sendcounts[k], sendtypes[k] with dsts[k]==j at process i must be equal to the type signature associated with recvcounts[l], recvtypes[l] with srcs[l]==i at process j. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of communicating processes. Distinct type maps between sender and receiver are still allowed.

The "in place" option is not meaningful for this operation.

All arguments are significant on all processes and the argument comm must have identical values on all processes.

## 7.7 Nonblocking Neighborhood Communication on Process Topologies

Nonblocking variants of the neighborhood collective operations allow relaxed synchronization and overlapping of computation and communication. The semantics are similar to nonblocking collective operations as described in Section 5.12.

### 7.7.1 Nonblocking Neighborhood Gather

MPI_INEIGHBOR_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | number of elements received from each neighbor (non-negative integer) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHER.

MPI_INEIGHBOR_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor |
| IN | displs | integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ineighbor_allgatherv(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
             displs, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLGATHERV.

### 7.7.2 Nonblocking Neighborhood Alltoall

MPI_INEIGHBOR_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | number of elements received from each neighbor (non-negative integer) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALL.

MPI_INEIGHBOR_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor |
| IN | sdispls | integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor |
| IN | rdispls | integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
             const int recvcounts[], const int rdispls[],
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

**F08 binding**

```
MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
             recvcounts, rdispls, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLV.

MPI_INEIGHBOR_ALLTOALLW(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor |
| IN | sdispls | integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers) |
| IN | sendtypes | array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor |
| IN | rdispls | integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers) |
| IN | recvtypes | array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles) |
| IN | comm | communicator with topology structure (handle) |
| OUT | request | communication request (handle) |

### C binding

```
int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[],
             const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
             void* recvbuf, const int recvcounts[],
             const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
             MPI_Comm comm, MPI_Request *request)
```

### F08 binding

```
MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
              sdispls(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
              REQUEST, IERROR
```

This call starts a nonblocking variant of MPI_NEIGHBOR_ALLTOALLW.

## 7.8 Persistent Neighborhood Communication on Process Topologies

Persistent variants of the neighborhood collective operations can offer significant performance benefits for programs with repetitive communication patterns. The semantics are similar to persistent collective operations as described in Section 5.13.

### 7.8.1 Persistent Neighborhood Gather

MPI_NEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | number of elements received from each neighbor (non-negative integer) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Neighbor_allgather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR
```

Creates a persistent collective communication request for the neighborhood allgather operation.

MPI_NEIGHBOR_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor |
| IN | displs | integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Neighbor_allgatherv_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcounts, displs, recvtype, comm, info, request, ierror)
```

```
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the neighborhood allgatherv operation.

### 7.8.2 Persistent Neighborhood Alltoall

MPI_NEIGHBOR_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each neighbor (non-negative integer) |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcount | number of elements received from each neighbor (non-negative integer) |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Neighbor_alltoall_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, info, request, ierror)
```

```
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR
```

Creates a persistent collective communication request for the neighborhood alltoall operation.

MPI_NEIGHBOR_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor |
| IN | sdispls | integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j |
| IN | sendtype | data type of send buffer elements (handle) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor |
| IN | rdispls | integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i |
| IN | recvtype | data type of receive buffer elements (handle) |
| IN | comm | communicator with topology structure (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

**C binding**

```
int MPI_Neighbor_alltoallv_init(const void* sendbuf,
             const int sendcounts[], const int sdispls[],
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
```

```
            const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
            MPI_Info info, MPI_Request *request)
```

**F08 binding**

```
MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
            recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
            ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
            RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
            IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the neighborhood alltoallv operation.

MPI_NEIGHBOR_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcounts | non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor |
| IN | sdispls | integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers) |
| IN | sendtypes | array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | recvcounts | non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor |
| IN | rdispls | integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers) |
| IN | recvtypes | array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles) |
| IN | comm | communicator with topology structure (handle) |
| IN | info | info argument (handle) |
| OUT | request | communication request (handle) |

### C binding

```
int MPI_Neighbor_alltoallw_init(const void* sendbuf,
             const int sendcounts[], const MPI_Aint sdispls[],
             const MPI_Datatype sendtypes[], void* recvbuf,
             const int recvcounts[], const MPI_Aint rdispls[],
             const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

### F08 binding

```
MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
             recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
              sdispls(*), rdispls(*)
```

```
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
             RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
              INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the neighborhood alltoallw operation.

## 7.9 An Application Example

**Example 7.9** The example in Figures 7.2-7.5 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 the points `u(1:100,1:100)` owned by the process. Then the values at inter-process boundaries have to be exchanged with neighboring processes. For example, the newly calculated values in `u(1,1:100)` must be sent into the halo cells `u(101,1:100)` of the left-hand neighbor with coordinates `(own_coord(1)-1,own_coord(2))`.

```
INTEGER ndims, num_neigh
LOGICAL reorder
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
INTEGER comm, comm_size, comm_cart, dims(ndims), ierr
INTEGER neigh_rank(num_neigh), own_coords(ndims), i, j, it
LOGICAL periods(ndims)
REAL u(0:101,0:101), f(0:101,0:101)
DATA dims / ndims * 0 /
comm = MPI_COMM_WORLD
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
!   Set process grid size and periodicity
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
periods(1) = .TRUE.
periods(2) = .TRUE.
!   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_coords, ierr)
i = own_coords(1)
j = own_coords(2)
! Look up the ranks for the neighbors.  Own process coordinates are (i,j).
! Neighbors are (i-1,j), (i+1,j), (i,j-1), (i,j+1) modulo (dims(1),dims(2))
CALL MPI_CART_SHIFT(comm_cart, 0,1, neigh_rank(1), neigh_rank(2), ierr)
CALL MPI_CART_SHIFT(comm_cart, 1,1, neigh_rank(3), neigh_rank(4), ierr)
! Initialize the grid functions and start the iteration
CALL init(u, f)
DO it=1,100
   CALL relax(u, f)
!      Exchange data with neighbor processes
   CALL exchange(u, comm_cart, neigh_rank, num_neigh)
END DO
CALL output(u)
```

Figure 7.2: Set-up of process structure for two-dimensional parallel Poisson solver.

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
INTEGER ierr
sndbuf(1:100,1) = u(  1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100,  1)
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL(sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                           comm_cart, ierr)
! instead of
! DO i=1,num_neigh
!   CALL MPI_IRECV(rcvbuf(1,i), 100, MPI_REAL, neigh_rank(i),..., &
!                  rq(2*i-1), ierr)
!   CALL MPI_ISEND(sndbuf(1,i), 100, MPI_REAL, neigh_rank(i),..., &
!                  rq(2*i  ), ierr)
! END DO
! CALL MPI_WAITALL(2*num_neigh, rq, statuses, ierr)

u(  0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
u(1:100,  0) = rcvbuf(1:100,3)
u(1:100,101) = rcvbuf(1:100,4)
END
```

Figure 7.3: Communication routine with local data copying and sparse neighborhood all-to-all.

```
SUBROUTINE exchange(u, comm_cart, neigh_rank, num_neigh)
IMPLICIT NONE
USE MPI
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
INTEGER type_vec, ierr
! The following initialization need to be done only once
! before the first call of exchange.
CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lb, sizeofreal, ierr)
CALL MPI_TYPE_VECTOR(100, 1, 102, MPI_REAL, type_vec, ierr)
CALL MPI_TYPE_COMMIT(type_vec, ierr)
sndtypes(1:2) = type_vec
sndcounts(1:2) = 1
sndtypes(3:4) = MPI_REAL
sndcounts(3:4) = 100
rcvtypes = sndtypes
rcvcounts = sndcounts
sdispls(1) = ( 1  +   1*102) * sizeofreal ! first element of u(  1    ,  1:100)
sdispls(2) = (100 +   1*102) * sizeofreal ! first element of u(100    ,  1:100)
sdispls(3) = ( 1  +   1*102) * sizeofreal ! first element of u(  1:100,  1    )
sdispls(4) = ( 1  + 100*102) * sizeofreal ! first element of u(  1:100,100    )
rdispls(1) = ( 0  +   1*102) * sizeofreal ! first element of u(  0    ,  1:100)
rdispls(2) = (101 +   1*102) * sizeofreal ! first element of u(101    ,  1:100)
rdispls(3) = ( 1  +   0*102) * sizeofreal ! first element of u(  1:100,  0    )
rdispls(4) = ( 1  + 101*102) * sizeofreal ! first element of u(  1:100,101    )
! the following communication has to be done in each call of exchange
CALL MPI_NEIGHBOR_ALLTOALLW(u, sndcounts, sdispls, sndtypes, &
                            u, rcvcounts, rdispls, rcvtypes, &
                            comm_cart, ierr)
! The following finalizing need to be done only once
! after the last call of exchange.
CALL MPI_TYPE_FREE(type_vec, ierr)
END
```

Figure 7.4: Communication routine with sparse neighborhood all-to-all-w and without local data copying.

```
INTEGER ndims, num_neigh
LOGICAL reorder
PARAMETER (ndims=2, num_neigh=4, reorder=.true.)
INTEGER comm, comm_size, comm_cart, dims(ndims), it, ierr
LOGICAL periods(ndims)
REAL u(0:101,0:101), f(0:101,0:101)
DATA dims / ndims * 0 /
INTEGER sndcounts(num_neigh), sndtypes(num_neigh)
INTEGER rcvcounts(num_neigh), rcvtypes(num_neigh)
INTEGER (KIND=MPI_ADDRESS_KIND) lb, sizeofreal
INTEGER (KIND=MPI_ADDRESS_KIND) sdispls(num_neigh), rdispls(num_neigh)
INTEGER type_vec, request, status
comm = MPI_COMM_WORLD
CALL MPI_COMM_SIZE(comm, comm_size, ierr)
!   Set process grid size and periodicity
CALL MPI_DIMS_CREATE(comm_size, ndims, dims, ierr)
periods(1) = .TRUE.
periods(2) = .TRUE.
!   Create a grid structure in WORLD group
CALL MPI_CART_CREATE(comm, ndims, dims, periods, reorder, &
                     comm_cart, ierr)
! Create datatypes for the neighborhood communication
!
! Insert code from example in Figure 7.4 to create and initialize
! sndcounts, sdispls, sndtypes, rcvcounts, rdispls, and rcvtypes
!
! Initialize the neighborhood all-to-all-w operation
CALL MPI_NEIGHBOR_ALLTOALLW_INIT(u, sndcounts, sdispls, sndtypes, &
                                 u, rcvcounts, rdispls, rcvtypes, &
                                 comm_cart, info, request, ierr)
! Initialize the grid functions and start the iteration
CALL init(u, f)
DO it=1,100
!      Start data exchange with neighbor processes
   CALL MPI_START(request, ierr)
!      Compute inner cells
   CALL relax_inner (u, f)
!      Check on completion of neighbor exchange
   CALL MPI_WAIT(request, status, ierr)
!      Compute edge cells
   CALL relax_edges(u, f)
END DO
CALL output(u)
CALL MPI_REQUEST_FREE(request, ierr)
CALL MPI_TYPE_FREE(type_vec, ierr)
```

Figure 7.5: Two-dimensional parallel Poisson solver with persistent sparse neighborhood all-to-all-w and without local data copying.

# 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 subversion: In C,

```
#define MPI_VERSION    3
#define MPI_SUBVERSION 1
```

in Fortran,

```
INTEGER :: MPI_VERSION, MPI_SUBVERSION
PARAMETER (MPI_VERSION    = 3)
PARAMETER (MPI_SUBVERSION = 1)
```

For runtime determination,

MPI_GET_VERSION(version, subversion)

| | | |
|---|---|---|
| OUT | version | version number (integer) |
| OUT | subversion | subversion number (integer) |

**C binding**

```
int MPI_Get_version(int *version, int *subversion)
```

**F08 binding**

```
MPI_Get_version(version, subversion, ierror)
```

```
    INTEGER, INTENT(OUT) :: version, subversion
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR
```

MPI_GET_VERSION can be called before MPI_INIT and after MPI_FINALIZE. This function must always be thread-safe, as defined in Section 12.4. Valid (MPI_VERSION, MPI_SUBVERSION) pairs in this and previous versions of the MPI standard are (3,1), (3,0), (2,2), (2,1), (2,0), and (1,2).

MPI_GET_LIBRARY_VERSION(version, resultlen)

| | | |
|---|---|---|
| OUT | version | version number (string) |
| OUT | resultlen | Length (in printable characters) of the result returned in version (integer) |

**C binding**

```
int MPI_Get_library_version(char *version, int *resultlen)
```

**F08 binding**

```
MPI_Get_library_version(version, resultlen, ierror)
    CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
    CHARACTER*(*) VERSION
    INTEGER RESULTLEN, IERROR
```

This routine returns a string representing the version of the MPI library. The version argument is a character string for maximum flexibility.

> *Advice to implementors.* An implementation of MPI should return a different string for every change to its source code or build that could be visible to the user. (*End of advice to implementors.*)

The argument version must represent storage that is MPI_MAX_LIBRARY_VERSION_STRING characters long. MPI_GET_LIBRARY_VERSION may write up to this many characters into version.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at version[resultlen]. The value of resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING - 1. In Fortran, version is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_LIBRARY_VERSION_STRING.

MPI_GET_LIBRARY_VERSION can be called before MPI_INIT and after MPI_FINALIZE. This function must always be thread-safe, as defined in Section 12.4.

### 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 values of these attributes can be inquired by using the function MPI_COMM_GET_ATTR described in Section 6.7 and in Section 17.2.7. 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_IO** 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_WTIME_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 valid 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 *HOST* 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 *HOST*, nor does it requires that a *HOST* exists.

The attribute MPI_HOST has the same value on all processes of MPI_COMM_WORLD.

#### IO Rank

The value returned for MPI_IO 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`, `REWIND`, `WRITE`). For C, this means that all of the ISO C I/O operations are supported (e.g., `fopen`, `fprintf`, `lseek`).

If every process can provide language-standard I/O, then the value MPI_ANY_SOURCE 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_GLOBAL is 1 if clocks at all processes in MPI_COMM_WORLD are synchronized, 0 otherwise. A collection of clocks is considered synchronized if explicit effort has been taken to synchronize them. The expectation is that the variation in time, as measured by calls to MPI_WTIME, will be less then one half the round-trip time for an MPI message of length zero. If time is measured at a process just before a send and at another process just after a matching receive, the second time should be always higher than the first one.

The attribute MPI_WTIME_IS_GLOBAL need not be present when the clocks are not synchronized (however, the attribute key MPI_WTIME_IS_GLOBAL is always valid). This attribute may be associated with communicators other then MPI_COMM_WORLD.

The attribute MPI_WTIME_IS_GLOBAL has the same value on all processes of MPI_COMM_WORLD.

### Inquire Processor Name

MPI_GET_PROCESSOR_NAME(name, resultlen)

| | | |
|---|---|---|
| OUT | name | A unique specifier for the actual (as opposed to virtual) node. (string) |
| OUT | resultlen | Length (in printable characters) of the result returned in name (integer) |

**C binding**

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

**F08 binding**

```
MPI_Get_processor_name(name, resultlen, ierror)
    CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) ::  name
    INTEGER, INTENT(OUT) ::  resultlen
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN, IERROR
```

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include "processor 9 in rack 4 of mpp.cs.org" and "231" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI_MAX_PROCESSOR_NAME characters long. MPI_GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen. In C, a null character is additionally stored at name[resultlen]. The value of resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. The value of resultlen cannot be larger than MPI_MAX_PROCESSOR_NAME.

> *Rationale.* This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI *requires* or defines process migration; this definition of MPI_GET_PROCESSOR_NAME simply allows such an implementation. (*End of rationale.*)

> *Advice to users.* The user must provide at least MPI_MAX_PROCESSOR_NAME space to write the processor name — processor names can be this long. The user should examine the output argument, resultlen, to determine the actual length of the name. (*End of advice to users.*)

## 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 some RMA functionality as defined in Section 11.5.3.

MPI_ALLOC_MEM(size, info, baseptr)

| | | |
|---|---|---|
| IN | size | size of memory segment in bytes (non-negative integer) |
| IN | info | info argument (handle) |
| OUT | baseptr | pointer to beginning of memory segment allocated |

**C binding**

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)
```

**F08 binding**

```
MPI_Alloc_mem(size, info, baseptr, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND, INTENT(IN) :: size
    TYPE(MPI_Info), INTENT(IN) :: info
```

```
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND SIZE
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR
```

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, but with a different specific procedure name:

```
INTERFACE MPI_ALLOC_MEM
    SUBROUTINE MPI_ALLOC_MEM(SIZE, INFO, BASEPTR, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER INFO, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_ALLOC_MEM_CPTR(SIZE, INFO, BASEPTR, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: INFO, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_ALLOC_MEM_CPTR. The implied specific procedure names are described in Section 17.1.5.

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.

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) |

**C binding**

```
int MPI_Free_mem(void *base)
```

**F08 binding**

```
MPI_Free_mem(base, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR
```

The function MPI_FREE_MEM may return an error code of class MPI_ERR_BASE to indicate an invalid base argument.

> *Rationale.* The 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 bindings: the Fortran MPI_ALLOC_MEM call returns in baseptr the `TYPE(C_PTR)` pointer or 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 `TYPE(C_PTR)` pointers. We assume 4-byte REALs.

```
USE mpi_f08   !  or  USE mpi     (not guaranteed with INCLUDE 'mpif.h')
USE, INTRINSIC :: ISO_C_BINDING
TYPE(C_PTR) :: p
REAL, DIMENSION(:,:), POINTER :: a            ! no memory is allocated
INTEGER, DIMENSION(2) :: shape
INTEGER(KIND=MPI_ADDRESS_KIND) :: size
shape = (/100,100/)
size = 4 * shape(1) * shape(2)                ! assuming 4 bytes per REAL
CALL MPI_Alloc_mem(size,MPI_INFO_NULL,p,ierr) ! memory is allocated and
CALL C_F_POINTER(p, a, shape) ! intrinsic     ! now accessible via a(i,j)
...                           ! in ISO_C_BINDING
a(3,5) = 2.71;
...
CALL MPI_Free_mem(a, ierr)                    ! memory is freed
```

**Example 8.2** Example of use of MPI_ALLOC_MEM, in Fortran with non-standard *Cray-pointers.* We assume 4-byte REALs, and assume that these pointers are address-sized.

```
REAL A
POINTER (P, A(100,100))   ! no memory is allocated
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
SIZE = 4*100*100
CALL MPI_ALLOC_MEM(SIZE, MPI_INFO_NULL, P, IERR)
! memory is allocated
...
A(3,5) = 2.71;
...
CALL MPI_FREE_MEM(A, IERR) ! memory is freed
```

This code is not Fortran 77 or Fortran 90 code. Some compilers may not support this code or need a special option, e.g., the GNU gFortran compiler needs `-fcray-pointer`.

> *Advice to implementors.* Some compilers map Cray-pointers to address-sized integers, some to `TYPE(C_PTR)` pointers (e.g., Cray Fortran, version 7.3.3). From the user's viewpoint, this mapping is irrelevant because Examples 8.2 should work correctly with an MPI-3.0 (or later) library if Cray-pointers are available. (*End of advice to implementors.*)

**Example 8.3** Same example, in 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. More background information about how MPI treats errors can be found in Section 2.8.

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_SELF. The attachment of error handlers to objects is purely local: different processes may attach different error handlers to corresponding objects.

Several predefined error handlers are available in MPI:

**MPI_ERRORS_ARE_FATAL** The handler, when called, causes the program to abort all connected processes. This is similar to calling MPI_ABORT using a communicator containing all connected processes with an implementation-specific value as the errorcode argument.

**MPI_ERRORS_ABORT** The handler, when called, is invoked on a communicator in a manner similar to calling MPI_ABORT on that communicator. If the error handler is invoked on an window or a file, it is similar to calling MPI_ABORT using a communicator containing the group of MPI processes associated with the window or file, respectively. In either case, the value that would be provided as the errorcode argument to MPI_ABORT is implementation-specific.

**MPI_ERRORS_RETURN** The handler has no effect other than returning the error code to the user.

*Advice to implementors.* The implementation-specific error information resulting from MPI_ERRORS_ARE_FATAL and MPI_ERRORS_ABORT provided to the invoking environment should be meaningful to the end-user, for example a predefined error class. (*End of advice to implementors.*)

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

After initialization, the error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI_COMM_WORLD, MPI_COMM_SELF, and the communicator returned by MPI_COMM_GET_PARENT (if any). 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, MPI will provide the user as much information as possible about that error using error classes. Some errors might prevent MPI from completing further API calls successfully and those functions will continue to report errors until the cause of the error is corrected or the user terminates the application. The user can make the determination of whether or not to attempt to continue after detecting such an error.

*Advice to users.* For example, users may be unable to correct errors corresponding to some error classes, such as MPI_ERR_INTERN. Such errors may cause subsequent MPI calls to complete in error. (*End of advice to users.*)

*Advice to implementors.* A high-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors and available recovery actions. (*End of advice to implementors.*)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with objects, and to test which error handler is associated with an object. C has distinct typedefs for user defined error handling callback functions that accept communicator, file, and window arguments. In Fortran there are three user routines.

An error handler object is created by a call to MPI_XXX_CREATE_ERRHANDLER, 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.

The error handler currently associated with a communicator, window, or file can be retrieved by a call to MPI_XXX_GET_ERRHANDLER.

The MPI function MPI_ERRHANDLER_FREE can be used to free an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER.

MPI_{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_{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 implementations should raise an error when an error handler that was created by a call to MPI_XXX_CREATE_ERRHANDLER is attached to an object of the wrong type with a call to MPI_YYY_SET_ERRHANDLER. To do so, it is necessary to maintain, with each error handler, information on the typedef of the associated user function. (*End of advice to implementors.*)

The syntax for these calls is given below.

### 8.3.1 Error Handlers for Communicators

# JMS See comment in LaTeX

MPI_COMM_CREATE_ERRHANDLER(comm_errhandler_fn, errhandler)

| | | |
|---|---|---|
| IN | comm_errhandler_fn | user defined error handling procedure (function) |
| OUT | errhandler | MPI error handler (handle) |

**C binding**

```
int
            MPI_Comm_create_errhandler(MPI_Comm_errhandler_function *comm_errhandler_fn,
            MPI_Errhandler *errhandler)
```

**F08 binding**

```
MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_Comm_errhandler_function) ::  comm_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) ::  errhandler
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL COMM_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
```

Creates an error handler that can be attached to communicators.

The user routine should be, in C, a function of type MPI_Comm_errhandler_function, which is defined as

```
typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine that raised the error. If the routine would have returned MPI_ERR_IN_STATUS, it is the error code returned in the status for the request that caused the error handler to be invoked. The remaining arguments are "varargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran. With the Fortran mpi_f08 module, the user routine comm_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
      TYPE(MPI_Comm) ::  comm
      INTEGER ::  error_code
```

With the Fortran mpi module and mpif.h, the user routine COMM_ERRHANDLER_FN should be of the form:

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
    INTEGER COMM, ERROR_CODE
```

*Rationale.* The variable argument list is provided because it provides an ISO-standard hook for providing additional information to the error handler; without this hook, ISO C prohibits additional arguments. (*End of rationale.*)

*Advice to users.* A newly created communicator inherits the error handler that is associated with the "parent" communicator. In particular, the user can specify 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) |

**C binding**

```
int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)
```

**F08 binding**

```
MPI_Comm_set_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
```

```
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_SET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

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.

MPI_COMM_GET_ERRHANDLER(comm, errhandler)

| | | |
|---|---|---|
| IN | comm | communicator (handle) |
| OUT | errhandler | error handler currently associated with communicator (handle) |

**C binding**

```
int MPI_Comm_get_errhandler(MPI_Comm comm, MPI_Errhandler *errhandler)
```

**F08 binding**

```
MPI_Comm_get_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_GET_ERRHANDLER(COMM, ERRHANDLER, IERROR)
    INTEGER COMM, ERRHANDLER, IERROR
```

Retrieves the error handler currently associated with a communicator.

For example, a library function may register at its entry point the current error handler for a communicator, set its own private error handler for this communicator, and restore before exiting the previous error handler.

### 8.3.2 Error Handlers for Windows

# JMS See comment in LaTeX

MPI_WIN_CREATE_ERRHANDLER(win_errhandler_fn, errhandler)

| | | |
|---|---|---|
| IN | win_errhandler_fn | user defined error handling procedure (function) |
| OUT | errhandler | MPI error handler (handle) |

**C binding**

```
int
            MPI_Win_create_errhandler(MPI_Win_errhandler_function *win_errhandler_fn,
            MPI_Errhandler *errhandler)
```

**F08 binding**

```
MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL WIN_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
```

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_function which is defined as

```
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
```

The first argument is the window in use, the second is the error code to be returned. With the Fortran `mpi_f08` module, the user routine win_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_errhandler_function(win, error_code)
      TYPE(MPI_Win) :: win
      INTEGER :: error_code
```

With the Fortran `mpi` module and `mpif.h`, the user routine WIN_ERRHANDLER_FN should be of the form:

```
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
    INTEGER WIN, ERROR_CODE
```

MPI_WIN_SET_ERRHANDLER(win, errhandler)

| | | |
|---|---|---|
| INOUT | win | window (handle) |
| IN | errhandler | new error handler for window (handle) |

**C binding**

```
int MPI_Win_set_errhandler(MPI_Win win, MPI_Errhandler errhandler)
```

**F08 binding**

```
MPI_Win_set_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_SET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

Attaches a new error handler to a window. The error handler must be either a predefined 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 (handle) |

**C binding**

```
int MPI_Win_get_errhandler(MPI_Win win, MPI_Errhandler *errhandler)
```

**F08 binding**

```
MPI_Win_get_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_GET_ERRHANDLER(WIN, ERRHANDLER, IERROR)
    INTEGER WIN, ERRHANDLER, IERROR
```

Retrieves the error handler currently associated with a window.

### 8.3.3 Error Handlers for Files

# JMS See comment in LaTeX

MPI_FILE_CREATE_ERRHANDLER(file_errhandler_fn, errhandler)

| | | |
|---|---|---|
| IN | file_errhandler_fn | user defined error handling procedure (function) |
| OUT | errhandler | MPI error handler (handle) |

**C binding**

```
int
        MPI_File_create_errhandler(MPI_File_errhandler_function *file_errhandler_fn,
        MPI_Errhandler *errhandler)
```

**F08 binding**

```
MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_CREATE_ERRHANDLER(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL FILE_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR
```

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_function, which is defined as

```
typedef void MPI_File_errhandler_function(MPI_File *, int *, ...);
```

The first argument is the file in use, the second is the error code to be returned. With the Fortran mpi_f08 module, the user routine file_errhandler_fn should be of the form:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
      TYPE(MPI_File) :: file
      INTEGER :: error_code
```

With the Fortran mpi module and mpif.h, the user routine FILE_ERRHANDLER_FN should be of the form:

```
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
    INTEGER FILE, ERROR_CODE
```

MPI_FILE_SET_ERRHANDLER(file, errhandler)

| | | |
|---|---|---|
| INOUT | file | file (handle) |
| IN | errhandler | new error handler for file (handle) |

**C binding**

```
int MPI_File_set_errhandler(MPI_File file, MPI_Errhandler errhandler)
```

**F08 binding**

```
MPI_File_set_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
```

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.

MPI_FILE_GET_ERRHANDLER(file, errhandler)

| | | |
|---|---|---|
| IN | file | file (handle) |
| OUT | errhandler | error handler currently associated with file (handle) |

**C binding**

```
int MPI_File_get_errhandler(MPI_File file, MPI_Errhandler *errhandler)
```

**F08 binding**

```
MPI_File_get_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
```

```
    INTEGER FILE, ERRHANDLER, IERROR
```

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) |

**C binding**

```
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
```

**F08 binding**

```
MPI_Errhandler_free(errhandler, ierror)
    TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
    INTEGER ERRHANDLER, IERROR
```

Marks the error handler associated with errhandler for deallocation and sets errhandler to MPI_ERRHANDLER_NULL. The error handler will be deallocated after all the objects associated with it (communicator, window, or file) have been deallocated.

MPI_ERROR_STRING(errorcode, string, resultlen)

| | | |
|---|---|---|
| IN | errorcode | Error code returned by an MPI routine (integer) |
| OUT | string | Text that corresponds to the errorcode (string) |
| OUT | resultlen | Length (in printable characters) of the result returned in string (integer) |

**C binding**

```
int MPI_Error_string(int errorcode, char *string, int *resultlen)
```

**F08 binding**

```
MPI_Error_string(errorcode, string, resultlen, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
    INTEGER ERRORCODE, RESULTLEN, IERROR
    CHARACTER*(*) STRING
```

Returns the error string associated with an error code or class. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long.

The number of characters actually written is returned in the output argument, resultlen.

*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. 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 (integer) |
| OUT | errorclass | Error class associated with errorcode (integer) |

**C binding**

```
int MPI_Error_class(int errorcode, int *errorclass)
```

**F08 binding**

```
MPI_Error_class(errorcode, errorclass, ierror)
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, INTENT(OUT) :: errorclass
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

| | |
|---|---|
| MPI_SUCCESS | No error |
| MPI_ERR_BUFFER | Invalid buffer pointer |
| MPI_ERR_COUNT | Invalid count argument |
| MPI_ERR_TYPE | Invalid datatype argument |
| MPI_ERR_TAG | Invalid tag argument |
| MPI_ERR_COMM | Invalid communicator |
| MPI_ERR_RANK | Invalid rank |
| MPI_ERR_REQUEST | Invalid request (handle) |
| MPI_ERR_ROOT | Invalid root |
| MPI_ERR_GROUP | Invalid group |
| MPI_ERR_OP | Invalid operation |
| MPI_ERR_TOPOLOGY | Invalid topology |
| MPI_ERR_DIMS | Invalid dimension argument |
| MPI_ERR_ARG | Invalid argument of some other kind |
| MPI_ERR_UNKNOWN | Unknown error |
| MPI_ERR_TRUNCATE | Message truncated on receive |
| MPI_ERR_OTHER | Known error not in this list |
| MPI_ERR_INTERN | Internal MPI (implementation) error |
| MPI_ERR_IN_STATUS | Error code is in status |
| MPI_ERR_PENDING | Pending request |
| MPI_ERR_KEYVAL | Invalid keyval has been passed |
| MPI_ERR_NO_MEM | MPI_ALLOC_MEM failed because memory is exhausted |
| MPI_ERR_BASE | Invalid base passed to MPI_FREE_MEM |
| MPI_ERR_INFO_KEY | Key longer than MPI_MAX_INFO_KEY |
| MPI_ERR_INFO_VALUE | Value longer than MPI_MAX_INFO_VAL |
| MPI_ERR_INFO_NOKEY | Invalid key passed to MPI_INFO_DELETE |
| MPI_ERR_SPAWN | Error in spawning processes |
| MPI_ERR_PORT | Invalid port name passed to MPI_COMM_CONNECT |
| MPI_ERR_SERVICE | Invalid service name passed to MPI_UNPUBLISH_NAME |
| MPI_ERR_NAME | Invalid service name passed to MPI_LOOKUP_NAME |
| MPI_ERR_WIN | Invalid win argument |
| MPI_ERR_SIZE | Invalid size argument |
| MPI_ERR_DISP | Invalid disp argument |
| MPI_ERR_INFO | Invalid info 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 8.1: Error classes (Part 1)

| | |
|---|---|
| MPI_ERR_RMA_RANGE | Target memory is not part of the window (in the case of a window created with MPI_WIN_CREATE_DYNAMIC, target memory is not attached) |
| MPI_ERR_RMA_ATTACH | Memory cannot be attached (e.g., because of resource exhaustion) |
| MPI_ERR_RMA_SHARED | Memory cannot be shared (e.g., some process in the group of the specified communicator cannot expose shared memory) |
| MPI_ERR_RMA_FLAVOR | Passed window has the wrong flavor for the called function |
| MPI_ERR_FILE | Invalid file handle |
| MPI_ERR_NOT_SAME | Collective argument not identical on all processes, or collective routines called in a different order by different processes |
| MPI_ERR_AMODE | Error related to the amode passed to MPI_FILE_OPEN |
| MPI_ERR_UNSUPPORTED_DATAREP | Unsupported datarep passed to MPI_FILE_SET_VIEW |
| MPI_ERR_UNSUPPORTED_OPERATION | Unsupported operation, such as seeking on a file which supports sequential access only |
| MPI_ERR_NO_SUCH_FILE | File does not exist |
| MPI_ERR_FILE_EXISTS | File exists |
| MPI_ERR_BAD_FILE | Invalid file name (e.g., path name too long) |
| MPI_ERR_ACCESS | Permission denied |
| MPI_ERR_NO_SPACE | Not enough space |
| MPI_ERR_QUOTA | Quota exceeded |
| MPI_ERR_READ_ONLY | Read-only file or file system |
| MPI_ERR_FILE_IN_USE | File operation could not be completed, as the file is currently open by some process |
| MPI_ERR_DUP_DATAREP | Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP |
| MPI_ERR_CONVERSION | An error occurred in a user supplied data conversion function. |
| MPI_ERR_IO | Other I/O error |
| MPI_ERR_LASTCODE | Last error code |

Table 8.2: Error classes (Part 2)

**F binding**

```
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
    INTEGER ERRORCODE, ERRORCLASS, IERROR
```

The function MPI_ERROR_CLASS maps each standard error code (error class) onto itself.

## 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. For this purpose, functions are needed to:

1. add a new error class to the ones an MPI implementation already knows.
2. associate error codes with this error class, so that MPI_ERROR_CLASS works.
3. associate strings with these error codes, so that MPI_ERROR_STRING works.
4. invoke the error handler associated with a communicator, window, or object.

Several functions are provided to do this. They are all local. No functions are provided to free error classes or codes: it is not expected that an application will generate them in significant numbers.

MPI_ADD_ERROR_CLASS(errorclass)

| | | |
|---|---|---|
| OUT | errorclass | value for the new error class (integer) |

**C binding**

```
int MPI_Add_error_class(int *errorclass)
```

**F08 binding**

```
MPI_Add_error_class(errorclass, ierror)
    INTEGER, INTENT(OUT) :: errorclass
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ADD_ERROR_CLASS(ERRORCLASS, IERROR)
    INTEGER ERRORCLASS, IERROR
```

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 errorclass 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.*)

MPI_ADD_ERROR_CODE(errorclass, errorcode)

| | | |
|---|---|---|
| IN | errorclass | error class (integer) |
| OUT | errorcode | new error code to be associated with errorclass (integer) |

**C binding**

```
int MPI_Add_error_code(int errorclass, int *errorcode)
```

**F08 binding**

```
MPI_Add_error_code(errorclass, errorcode, ierror)
    INTEGER, INTENT(IN) :: errorclass
    INTEGER, INTENT(OUT) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ADD_ERROR_CODE(ERRORCLASS, ERRORCODE, IERROR)
    INTEGER ERRORCLASS, ERRORCODE, IERROR
```

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) |

**C binding**

```
int MPI_Add_error_string(int errorcode, const char *string)
```

**F08 binding**

```
MPI_Add_error_string(errorcode, string, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=*), INTENT(IN) :: string
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ADD_ERROR_STRING(ERRORCODE, STRING, IERROR)
    INTEGER ERRORCODE, IERROR
    CHARACTER*(*) 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 as defined in the calling language. The length of the string does not include the null terminator in 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).

Section 8.3 describes the methods for creating and associating error handlers with communicators, files, and windows.

MPI_COMM_CALL_ERRHANDLER(comm, errorcode)

| | | |
|---|---|---|
| IN | comm | communicator with error handler (handle) |
| IN | errorcode | error code (integer) |

**C binding**

```
int MPI_Comm_call_errhandler(MPI_Comm comm, int errorcode)
```

**F08 binding**

```
MPI_Comm_call_errhandler(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

This function invokes the error handler assigned to the communicator with the error code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

MPI_WIN_CALL_ERRHANDLER(win, errorcode)

| | | |
|---|---|---|
| IN | win | window with error handler (handle) |
| IN | errorcode | error code (integer) |

**C binding**

```
int MPI_Win_call_errhandler(MPI_Win win, int errorcode)
```

**F08 binding**

```
MPI_Win_call_errhandler(win, errorcode, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR
```

This function invokes the error handler assigned to the window with the error code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

> *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) |

**C binding**

```
int MPI_File_call_errhandler(MPI_File fh, int errorcode)
```

**F08 binding**

```
MPI_File_call_errhandler(fh, errorcode, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_CALL_ERRHANDLER(FH, ERRORCODE, IERROR)
    INTEGER FH, ERRORCODE, IERROR
```

This function invokes the error handler assigned to the file with the error code supplied. This function returns MPI_SUCCESS in C and the same value in IERROR if the error handler was successfully called (assuming the process is not aborted and the error handler returns).

*Advice to users.* 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.*)

## 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 2.6.4.

MPI_WTIME()

**C binding**

```
double MPI_Wtime(void)
```

**F08 binding**

```
DOUBLE PRECISION MPI_Wtime()
```

**F binding**

```
DOUBLE PRECISION MPI_WTIME()
```

MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
    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 in Section 8.1.2).

MPI_WTICK()

**C binding**

`double MPI_Wtick(void)`

**F08 binding**

`DOUBLE PRECISION MPI_Wtick()`

**F binding**

`DOUBLE PRECISION 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 binding**

`int MPI_Init(int *argc, char ***argv)`

**F08 binding**

```
MPI_Init(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_INIT(IERROR)
    INTEGER IERROR
```

All MPI programs must contain exactly one call to an MPI initialization routine: MPI_INIT or MPI_INIT_THREAD. Subsequent calls to any initialization routines are erroneous. The only MPI functions that may be invoked before the MPI initialization routines are called are MPI_GET_VERSION, MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED, MPI_FINALIZED, and any function with the prefix MPI_T_ (within the constraints for functions with this prefix listed in Section ??). The version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program    */

    MPI_Finalize();     /* see below */
    return 0;
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc and argv arguments of main in C.

After MPI is initialized, the application can access information about the execution environment by querying the predefined info object MPI_INFO_ENV. The following keys are predefined for this object, corresponding to the arguments of MPI_COMM_SPAWN or of mpiexec:

command Name of program executed.

argv Space separated arguments to command.

maxprocs Maximum number of MPI processes to start.

soft Allowed values for number of processors.

host Hostname.

arch Architecture name.

wdir Working directory of the MPI process.

file Value is the name of a file in which additional information is specified.

thread_level Requested level of thread support, if requested before the program started execution.

Note that all values are strings. Thus, the maximum number of processes is represented by a string such as "1024" and the requested level is represented by a string such as "MPI_THREAD_SINGLE".

The info object MPI_INFO_ENV need not contain a (key,value) pair for each of these predefined keys; the set of (key,value) pairs provided is implementation-dependent. Implementations may provide additional, implementation specific, (key,value) pairs.

In case where the MPI processes were started with MPI_COMM_SPAWN_MULTIPLE or, equivalently, with a startup mechanism that supports multiple process specifications, then the values stored in the info object MPI_INFO_ENV at a process are those values that affect the local MPI process.

**Example 8.4** If MPI is started with a call to

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

Then the first 5 processes will have have in their MPI_INFO_ENV object the pairs (command, ocean), (maxprocs, 5), and (arch, sun). The next 10 processes will have in MPI_INFO_ENV (command, atmos), (maxprocs, 10), and (arch, rs6000)

> *Advice to users.* The values passed in MPI_INFO_ENV are the values of the arguments passed to the mechanism that started the MPI execution — not the actual value provided. Thus, the value associated with maxprocs is the number of MPI processes requested; it can be larger than the actual number of processes obtained, if the soft option was used. (*End of advice to users.*)

> *Advice to implementors.* High-quality implementations will provide a (key,value) pair for each parameter that can be passed to the command that starts an MPI program. (*End of advice to implementors.*)

MPI_FINALIZE()

**C binding**

```
int MPI_Finalize(void)
```

**F08 binding**

```
MPI_Finalize(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

This routine cleans up all MPI state. If an MPI program terminates normally (i.e., not due to a call to MPI_ABORT or an unrecoverable error) then each process must call MPI_FINALIZE before it exits.

Before an MPI process invokes MPI_FINALIZE, the process must perform all MPI calls needed to complete its involvement in MPI communications: It must locally complete all MPI operations that it initiated and must execute matching calls needed to complete MPI communications initiated by other processes. For example, if the process executed a non-blocking send, it must eventually call MPI_WAIT, MPI_TEST, MPI_REQUEST_FREE, or any derived function; if the process is the target of a send, then it must post the matching receive; if it is part of a group executing a collective operation, then it must have completed its participation in the operation.

The call to MPI_FINALIZE does not free objects created by MPI calls; these objects are freed using MPI_XXX_FREE calls.

MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4.

The following examples illustrates these rules

**Example 8.5** The following code is correct

```
        Process 0                Process 1
        ---------                ---------
        MPI_Init();              MPI_Init();
        MPI_Send(dest=1);        MPI_Recv(src=0);
        MPI_Finalize();          MPI_Finalize();
```

**Example 8.6** Without a matching receive, the program is erroneous

```
        Process 0                Process 1
        -----------              -----------
        MPI_Init();              MPI_Init();
        MPI_Send (dest=1);
        MPI_Finalize();          MPI_Finalize();
```

**Example 8.7** This program is correct: Process 0 calls MPI_Finalize after it has executed the MPI calls that complete the send operation. Likewise, process 1 executes the MPI call that completes the matching receive operation before it calls MPI_Finalize.

```
  Process 0                     Proces 1
  --------                      --------
  MPI_Init();                   MPI_Init();
  MPI_Isend(dest=1);            MPI_Recv(src=0);
  MPI_Request_free();           MPI_Finalize();
  MPI_Finalize();               exit();
exit();
```

**Example 8.8** This program is correct. The attached buffer is a resource allocated by the user, not by MPI; it is available to the user after MPI is finalized.

```
   Process 0                     Process 1
   ---------                     ---------
   MPI_Init();                  MPI_Init();
   buffer = malloc(1000000);    MPI_Recv(src=0);
   MPI_Buffer_attach();         MPI_Finalize();
   MPI_Send(dest=1));           exit();
   MPI_Finalize();
   free(buffer);
   exit();
```

**Example 8.9** This program is correct. The cancel operation must succeed, since the send cannot complete normally. The wait operation, after the call to MPI_Cancel, is local — no matching MPI call is required on process 1. Cancelling a send request by calling MPI_CANCEL is deprecated.

```
Process 0                    Process 1
---------                    ---------
MPI_Issend(dest=1);          MPI_Finalize();
MPI_Cancel();
MPI_Wait();
MPI_Finalize();
```

*Advice to implementors.* Even though a process has executed all MPI calls needed to complete the communications it is involved with, such communication may not yet be completed from the viewpoint of the underlying MPI system. For example, a blocking send may have returned, even though the data is still buffered at the sender in an MPI buffer; an MPI process may receive a cancel request for a message it has completed receiving. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. The MPI implementation should also complete freeing all objects marked for deletion by MPI calls that freed them. (*End of advice to implementors.*)

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except for MPI_GET_VERSION, MPI_GET_LIBRARY_VERSION, MPI_INITIALIZED, MPI_FINALIZED, and any function with the prefix MPI_T_ (within the constraints for functions with this prefix listed in Section ??).

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, users may desire to supply an exit code for each process that returns from MPI_FINALIZE.

**Example 8.10** The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.

```
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
...
MPI_Finalize();
if (myrank == 0) {
    resultfile = fopen("outfile", "w");
    dump_results(resultfile);
    fclose(resultfile);
}
exit(0);
```

MPI_INITIALIZED(flag)

| | | |
|---|---|---|
| OUT | flag | Flag is true if MPI_INIT has been called and flags otherwise (logical) |

**C binding**

```
int MPI_Initialized(int *flag)
```

**F08 binding**

```
MPI_Initialized(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

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. This function must always be thread-safe, as defined in Section 12.4.

MPI_ABORT(comm, errorcode)

| | | |
|---|---|---|
| IN | comm | communicator of tasks to abort (handle) |
| IN | errorcode | error code to return to invoking envionment (integer) |

**C binding**

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

**F08 binding**

```
MPI_Abort(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_ABORT(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR
```

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.

> *Advice to implementors.* After aborting a subset of processes, a high quality implementation should be able to provide error handling for communicators, windows, and files involving both aborted and non-aborted processes. As an example, if the user changes the error handler for MPI_COMM_WORLD to MPI_ERRORS_RETURN or a custom error handler, when a subset of MPI_COMM_WORLD is aborted, the remaining processes in MPI_COMM_WORLD should be able to continue communicating with each other and receive appropriate error codes when attempting communication with an aborted process. (*End of advice to implementors.*)

> *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.*)

### 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 the reverse order that they were set on MPI_COMM_SELF. If no key has been attached to MPI_COMM_SELF, then no callback is invoked. The "freeing" of MPI_COMM_SELF occurs before any other parts of MPI are affected. Thus, for example, calling MPI_FINALIZED will return false in any of these callback functions. Once done with MPI_COMM_SELF, the order and rest of the actions taken by MPI_FINALIZE is not specified.

> *Advice to implementors.* Since attributes can be added from any supported language, the MPI implementation needs to remember the creating language so the correct callback is made. Implementations that use the attribute delete callback on MPI_COMM_SELF internally should register their internal callbacks before returning from MPI_INIT / MPI_INIT_THREAD, so that libraries or applications will not have portions of the MPI implementation shut down before the application-level callbacks are made. (*End of advice to implementors.*)

### 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:

MPI_FINALIZED(flag)

| | | |
|---|---|---|
| OUT | flag | true if MPI was finalized (logical) |

**C binding**

```
int MPI_Finalized(int *flag)
```

**F08 binding**

```
MPI_Finalized(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

This routine returns true if MPI_FINALIZE has completed. It is valid to call MPI_FINALIZED before MPI_INIT and after MPI_FINALIZE. This function must always be thread-safe, as defined in Section 12.4.

> *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 MPI program itself.

Having a standard startup mechanism also extends the portability of MPI programs one step further, to the command lines and scripts that manage them. For example, a validation suite script that runs hundreds of programs can be a portable script if it is written using such a standard starup mechanism. In order that the "standard" command not be confused with existing practice, which is not standard and not portable among implementations, instead of `mpirun` MPI specifies `mpiexec`.

While a standardized startup mechanism improves the usability of MPI, the range of environments is so diverse (e.g., there may not even be a command line interface) that MPI cannot mandate such a mechanism. Instead, MPI specifies an `mpiexec` startup command

and recommends but does not require it, as advice to implementors. However, if an implementation does provide a command called `mpiexec`, it must be of the form described below.

It is suggested that

```
mpiexec -n <numprocs> <program>
```

be at least one way to start `<program>` with an initial MPI_COMM_WORLD whose group contains `<numprocs>` processes. Other arguments to `mpiexec` may be implementation-dependent.

*Advice to implementors.* Implementors, if they do provide a special startup command for MPI programs, are advised to give it the following form. The syntax is chosen in order that `mpiexec` be able to be viewed as a command-line version of MPI_COMM_SPAWN (See Section 10.3.4).

Analogous to MPI_COMM_SPAWN, we have

```
mpiexec -n    <maxprocs>
       -soft  <        >
       -host  <        >
       -arch  <        >
       -wdir  <        >
       -path  <        >
       -file  <        >
        ...
       <command line>
```

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 MPI_COMM_SPAWN_MULTIPLE there are two possible formats:

Form A:

```
mpiexec { <above arguments> } : { ... } : { ... } : ... : { ... }
```

As with MPI_COMM_SPAWN, all the arguments are optional. (Even the `-n 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 info argument to 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 8.11** Start 16 instances of `myprog` on the current or default machine:

```
mpiexec -n 16 myprog
```

**Example 8.12** Start 10 processes on the machine called `ferrari`:

```
mpiexec -n 10 -host ferrari myprog
```

**Example 8.13** Start three copies of the same program with different command-line arguments:

```
mpiexec myprog infile1 : myprog infile2 : myprog infile3
```

**Example 8.14** Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's:

```
mpiexec -n 5 -arch sun ocean : -n 10 -arch rs6000 atmos
```

It is assumed that the implementation in this case has a method for choosing hosts of the appropriate type. Their ranks are in the order specified.

**Example 8.15** Start the `ocean` program on five Suns and the `atmos` program on 10 RS/6000's (Form B):

```
mpiexec -configfile myfile
```

where `myfile` contains

```
-n 5  -arch sun    ocean
-n 10 -arch rs6000 atmos
```

(*End of advice to implementors.*)

# 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 and Fortran with the mpi_f08 module, and INTEGER in Fortran with the mpi module or the include file mpif.h. It stores an unordered set of (key,value) pairs (both key and value are strings). A key can have only one value. MPI reserves several keys and requires that if an implementation uses a reserved key, it must provide the specified functionality. An implementation is not required to support these keys and may support any others not reserved by MPI. Some info hints allow the MPI library to restrict its support for certain operations in order to improve performance or resource utilization. If an application provides such an info hint, it must be compatible with any changes in the behavior of the MPI library that are allowed by the info hint.

An implementation must support info objects as caches for arbitrary (key,value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, 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 info is used as an argument to a nonblocking routine, it 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. Valid values for a boolean must include the strings "true" and "false" (all lowercase). For integers, valid 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 integer is a valid 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 valid 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) |

**C binding**

```
int MPI_Info_create(MPI_Info *info)
```

**F08 binding**

```
MPI_Info_create(info, ierror)
    TYPE(MPI_Info), INTENT(OUT) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_CREATE(INFO, IERROR)
    INTEGER INFO, IERROR
```

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) |

**C binding**

```
int MPI_Info_set(MPI_Info info, const char *key, const char *value)
```

**F08 binding**

```
MPI_Info_set(info, key, value, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key, value
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
    INTEGER INFO, IERROR
```

```
    CHARACTER*(*) KEY, 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) |

**C binding**

```
int MPI_Info_delete(MPI_Info info, const char *key)
```

**F08 binding**

```
MPI_Info_delete(info, key, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_DELETE(INFO, KEY, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) 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 (logical) |

**C binding**

```
int MPI_Info_get(MPI_Info info, const char *key, int valuelen, char *value,
            int *flag)
```

**F08 binding**

```
MPI_Info_get(info, key, valuelen, value, flag, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, INTENT(IN) :: valuelen
    CHARACTER(LEN=*), INTENT(OUT) :: value
```

```
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    CHARACTER*(*) KEY, VALUE
    LOGICAL FLAG
```

This function retrieves the value associated with key in a previous call to MPI_INFO_SET. If such a key exists, it sets flag to true and returns the value in value, otherwise it sets flag to false and leaves value unchanged. 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, flag)

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

**C binding**

```
int MPI_Info_get_valuelen(MPI_Info info, const char *key, int *valuelen,
             int *flag)
```

**F08 binding**

```
MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, INTENT(OUT) :: valuelen
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
    INTEGER INFO, VALUELEN, IERROR
    CHARACTER*(*) KEY
    LOGICAL FLAG
```

Retrieves the length of the value associated with key. If key is defined, valuelen is set to the length of its associated value and flag is set to true. If key is not defined, valuelen is not touched and flag is set to false. The length returned in C does not include the end-of-string character.

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) |

**C binding**

```
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
```

**F08 binding**

```
MPI_Info_get_nkeys(info, nkeys, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(OUT) :: nkeys
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
    INTEGER INFO, NKEYS, IERROR
```

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) |

**C binding**

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
```

**F08 binding**

```
MPI_Info_get_nthkey(info, n, key, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: n
    CHARACTER(LEN=*), INTENT(OUT) :: key
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
    INTEGER INFO, N, IERROR
    CHARACTER*(*) KEY
```

This function returns the nth defined key in info. Keys are numbered $0 \ldots N-1$ 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) |

**C binding**

```
int MPI_Info_dup(MPI_Info info, MPI_Info *newinfo)
```

**F08 binding**

```
MPI_Info_dup(info, newinfo, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Info), INTENT(OUT) :: newinfo
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_DUP(INFO, NEWINFO, IERROR)
    INTEGER INFO, NEWINFO, IERROR
```

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) |

**C binding**

```
int MPI_Info_free(MPI_Info *info)
```

**F08 binding**

```
MPI_Info_free(info, ierror)
    TYPE(MPI_Info), INTENT(INOUT) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INFO_FREE(INFO, IERROR)
    INTEGER INFO, IERROR
```

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 allows 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 intial 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 latter form of process creation comes from the PVM [24] 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. MPI 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 presense 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.

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

### 10.2.2 The Runtime Environment

The MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE routines provide an interface between MPI and the *runtime environment* of an MPI application. The difficulty is that there is an enormous range of runtime environments and application requirements, and MPI must not be tailored to any particular one. Examples of such environments are:

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

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

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

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

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 environment-specific info are not portable.

MPI does not require the existence of an underlying "virtual machine" model, in which there is a consistent global view of an MPI application and an implicit "operating system" managing resources and processes. For instance, processes spawned by one task may not be visible to another; additional hosts added to the runtime environment by one process may not be visible in another process; tasks spawned by different processes may not be automatically distributed over available resources.

Interaction between MPI and the runtime environment is limited to the following areas:

- A process may start new processes with MPI_COMM_SPAWN and MPI_COMM_SPAWN_MULTIPLE.

- When a process spawns a child process, it may optionally use an info argument to tell the runtime environment where or how to start the process. This extra information may be opaque to MPI.

- An attribute MPI_UNIVERSE_SIZE (See Section 10.5.1) on MPI_COMM_WORLD tells a program how "large" the initial runtime environment is, namely how many processes can usefully be started in all. One can subtract the size of MPI_COMM_WORLD from this value to find out how many processes might usefully be started in addition to those already running.

## 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 first starting one process and having that process start its siblings with MPI_COMM_SPAWN. This practice is discouraged primarily for reasons of performance. If possible, it is preferable to start all processes at once, as a single MPI application. (*End of advice to users.*)

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, 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 (non-negative 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) |

**C binding**

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

**F08 binding**

```
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
             array_of_errcodes, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  command, argv(*)
    INTEGER, INTENT(IN) ::  maxprocs, root
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  intercomm
    INTEGER ::  array_of_errcodes(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_SPAWN(COMMAND, ARGV, MAXPROCS, INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERROR)
    CHARACTER*(*) COMMAND, ARGV(*)
    INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERROR
```

MPI_COMM_SPAWN tries to start maxprocs identical copies of the MPI program 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 sys-

tem 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 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 an empty string in Fortran. In Fortran, leading and trailing spaces are always stripped, so that a string consisting of all spaces is considered an empty string. The constant MPI_ARGV_NULL may be used in C and Fortran to indicate an empty argument list. In C this constant is the same as NULL.

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

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

or, if not everything is known at compile time:

```
char *command;
char **argv;
command = "ocean";
argv=(char **)malloc(3 * sizeof(char *));
argv[0] = "-gridfile";
argv[1] = "ocean1.grd";
argv[2] = NULL;
MPI_Comm_spawn(command, argv, ...);
```

In Fortran:

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

Arguments are supplied to the program if this is allowed by the operating system. In C, the MPI_COMM_SPAWN argument argv differs from the argv argument of main in two respects. First, it is shifted by one element. Specifically, argv[0] of main is provided by the implementation and conventionally contains the name of the program (given by command). argv[1] of main corresponds to argv[0] in MPI_COMM_SPAWN, argv[2] of main to argv[1] of MPI_COMM_SPAWN, etc. Passing an argv of MPI_ARGV_NULL to MPI_COMM_SPAWN results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program. Second, argv of MPI_COMM_SPAWN must be null-terminated, so that its length can be determined.

If a Fortran implementation supplies routines that allow a program to obtain its arguments, the arguments may be available through that mechanism. In C, if the operating system does not support arguments appearing in argv of main(), the MPI implementation may add the arguments to the argv that is passed to MPI_INIT.

**The maxprocs argument** MPI tries to spawn maxprocs processes. If it is unable to spawn maxprocs processes, it raises an error of class MPI_ERR_SPAWN.

An implementation may allow the info argument to change the default behavior, such that if the implementation is unable to spawn all maxprocs processes, it may spawn a smaller number of processes instead of raising an error. In principle, the info argument may specify an arbitrary set $\{m_i : 0 \leq m_i \leq \text{maxprocs}\}$ of allowed values for the number of processes spawned. The set $\{m_i\}$ does not necessarily include the value maxprocs. If an implementation is able to spawn one of these allowed numbers of processes, MPI_COMM_SPAWN returns successfully and the number of spawned processes, $m$, is given by the size of the remote group of intercomm. If $m$ is less than maxproc, reasons why the other processes were not spawned are given in array_of_errcodes 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 for more information on the soft key for info.

> *Advice to users.* By default, requests are hard and MPI errors are fatal. This means that by default there will be a fatal error if MPI cannot spawn all the requested processes. If you want the behavior "spawn as many processes as possible, up to $N$," you should do a soft spawn, where the set of allowed values $\{m_i\}$ is $\{0 \ldots N\}$. However, this is not completely portable, as implementations are not required to support soft spawning. (*End of advice to users.*)

**The info argument** The info argument to all of the routines in this chapter is an opaque handle of type MPI_Info in C and Fortran with the mpi_f08 module and INTEGER in Fortran with the mpi module or the include file mpif.h. It 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 Chapter 9.

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). 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 $m$ ($0 \le m <$ maxprocs) processes are spawned, $m$ of the entries will contain MPI_SUCCESS and the rest will contain an implementation-specific error code indicating the reason MPI could not start the process. MPI does not specify which entries correspond to failed processes. An implementation may, for instance, fill in error codes in one-to-one correspondence with a detailed specification in the info argument. These error codes all belong to the error class MPI_ERR_SPAWN if there was no error in the argument list. In C or Fortran, an application may pass MPI_ERRCODES_IGNORE if it is not interested in the error codes.

> *Advice to implementors.* MPI_ERRCODES_IGNORE in Fortran is a special type of constant, like MPI_BOTTOM. See the discussion in Section 2.5.4. (*End of advice to implementors.*)

MPI_COMM_GET_PARENT(parent)

| | | |
|---|---|---|
| OUT | parent | the parent communicator (handle) |

**C binding**

```
int MPI_Comm_get_parent(MPI_Comm *parent)
```

**F08 binding**

```
MPI_Comm_get_parent(parent, ierror)
    TYPE(MPI_Comm), INTENT(OUT) :: parent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_GET_PARENT(PARENT, IERROR)
```

```
INTEGER PARENT, IERROR
```

If a process was started with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, MPI_COMM_GET_PARENT returns the "parent" 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.

MPI_COMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes)

| | | |
|---|---|---|
| IN | count | number of commands (integer, significant only at root) |
| IN | array_of_commands | programs to be executed (array of strings, significant only at root) |
| IN | array_of_argv | arguments for commands (array of array of array of stringss, significant only at root) |
| IN | array_of_maxprocs | maximum number of processes to start for each command (non-negative integer, significant only at root) |
| IN | array_of_info | info objects telling the runtime system where and how to start processes (handle, significant only at root) |
| IN | root | rank of process in which previous arguments are examined (non-negative 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 (integer) |

**C binding**

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

**F08 binding**

```
MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
             array_of_maxprocs, array_of_info, root, comm, intercomm,
             array_of_errcodes, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_maxprocs(*), root
    CHARACTER(LEN=*), INTENT(IN) ::  array_of_commands(*),
    array_of_argv(count,*)
    TYPE(MPI_Info), INTENT(IN) ::  array_of_info(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  intercomm
    INTEGER ::  array_of_errcodes(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERROR)
    INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(*)
```

MPI_COMM_SPAWN_MULTIPLE is identical to MPI_COMM_SPAWN except that there are multiple executable specifications. The first argument, count, gives the number of specifications. Each of the next four arguments are simply arrays of the corresponding arguments in MPI_COMM_SPAWN. For the Fortran version of array_of_argv, the element array_of_argv(i,j) is the j-th argument to command number i.

*Rationale.* This may seem backwards to Fortran programmers who are familiar with Fortran's column-major ordering. However, it is necessary to do it this way to allow MPI_COMM_SPAWN to sort out arguments. Note that the leading dimension of array_of_argv *must* be the same as count. Also note that Fortran rules for sequence association allow a different value in the first dimension; in this case, the sequence of array elements is interpreted by MPI_COMM_SPAWN_MULTIPLE as if the sequence is stored in an array defined with the first dimension set to count. This Fortran feature allows an implementor to define MPI_ARGVS_NULL (see below) with fixed dimensions, e.g., (1,1), or only with one dimension, e.g., (1). (*End of rationale.*)

*Advice to users.* The argument count is interpreted by MPI only at the root, as is array_of_argv. Since the leading dimension of array_of_argv is count, a non-positive value of count at a non-root node could theoretically cause a runtime bounds check error, even though array_of_argv should be ignored by the subroutine. If this happens, you should explicitly supply a reasonable value of count on the non-root nodes. (*End of advice to users.*)

In any language, an application may use the constant MPI_ARGVS_NULL (which is likely to be `(char ***)0` in C) to specify that no arguments should be passed to any commands. The effect of setting individual elements of array_of_argv to MPI_ARGV_NULL is not defined. To specify arguments for some commands but not others, the commands without arguments should have a corresponding argv whose first element is null (`(char *)0` in C and empty string in Fortran). In Fortran at non-root processes, the count argument must be set to a value that is consistent with the provided array_of_argv although the content of these arguments has no meaning for this operation.

All of the spawned processes have the same MPI_COMM_WORLD. Their ranks in MPI_COMM_WORLD correspond directly to the order in which the commands are specified in MPI_COMM_SPAWN_MULTIPLE. Assume that $m_1$ processes are generated by the first command, $m_2$ by the second, etc. The processes corresponding to the first command have ranks $0, 1, \ldots, m_1-1$. The processes in the second command have ranks $m_1, m_1+1, \ldots, m_1+m_2-1$. The processes in the third have ranks $m_1+m_2, m_1+m_2+1, \ldots, m_1+m_2+m_3-1$, etc.

*Advice to users.* Calling MPI_COMM_SPAWN multiple times would create many sets of children with different MPI_COMM_WORLDs whereas MPI_COMM_SPAWN_MULTIPLE creates children with a single MPI_COMM_WORLD, 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 $\sum_{i=1}^{count} n_i$, where $n_i$ is the $i$-th element of array_of_maxprocs. Command number $i$ corresponds to the $n_i$ contiguous slots in this array from element $\sum_{j=1}^{i-1} n_j$ to $\left[\sum_{j=1}^{i} n_j\right] - 1$. Error codes are treated 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:

```
char *array_of_commands[2] = {"ocean", "atmos"};
char **array_of_argv[2];
char *argv0[] = {"-gridfile", "ocean1.grd", (char *)0};
char *argv1[] = {"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 is how you do it in 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) = ' '

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.

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 $a, a+1, a+2, \ldots, b$
3. a:b:c means $a, a+c, a+2c, \ldots, a+ck$, where for $c > 0$, $k$ is the largest integer for which $a + ck \leq b$ and for $c < 0$, $k$ is the largest integer for which $a + ck \geq b$. If $b > a$ then $c$ must be positive. If $b < a$ then $c$ must be negative.

Examples:

1. a:b gives a range between $a$ and $b$
2. 0:N gives full "soft" functionality
3. 1,2,4,8,16,32,64,128,256,512,1024,2048,4096 allows a power-of-two number of processes.
4. 2:10000:2 allows an even number of processes.
5. 2:10:2,7 allows 2, 4, 6, 7, 8, or 10 processes.

### 10.3.5 Spawn Example

#### Manager-worker Example Using MPI_COMM_SPAWN

```
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
   int world_size, universe_size, *universe_sizep, flag;
   MPI_Comm everyone;           /* intercommunicator */
   char worker_program[100];

   MPI_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\n\
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");

   /*
    * 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 does not 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.

Agreeing on a rendezvous point always involves a third party. The third party may itself provide the rendezvous point or may communicate rendezvous information from server to client. Complicating matters might be the fact that a client does not really care what server it contacts, only that it be able to get in touch with one that can handle its request.

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) |

**C binding**

```
int MPI_Open_port(MPI_Info info, char *port_name)
```

**F08 binding**

```
MPI_Open_port(info, port_name, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) 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) |

**C binding**

```
int MPI_Close_port(const char *port_name)
```

**F08 binding**

```
MPI_Close_port(port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_CLOSE_PORT(PORT_NAME, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER IERROR
```

MPI_COMM_ACCEPT(port_name, info, root, comm, newcomm)

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

**C binding**

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

**F08 binding**

```
MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_ACCEPT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

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 can be used to provide directives that may influence the behavior of 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, significant only at root) |
| IN | info | implementation-dependent information (handle, significant only at root) |
| IN | root | rank in comm of root node (non-negative integer) |
| IN | comm | intracommunicator over which call is collective (handle) |
| OUT | newcomm | intercommunicator with server as remote group (handle) |

**C binding**

```
int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
             MPI_Comm comm, MPI_Comm *newcomm)
```

**F08 binding**

```
MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  port_name
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, INTENT(IN) ::  root
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMM_CONNECT(PORT_NAME, INFO, ROOT, COMM, NEWCOMM, IERROR)
    CHARACTER*(*) PORT_NAME
    INTEGER INFO, ROOT, COMM, NEWCOMM, IERROR
```

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

### 10.4.4 Name Publishing

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

MPI_PUBLISH_NAME(service_name, info, port_name)

| | | |
|---|---|---|
| IN | service_name | a service name to associate with the port (string) |
| IN | info | implementation-specific information (handle) |
| IN | port_name | a port name (string) |

**C binding**

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

**F08 binding**

```
MPI_Publish_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  service_name, port_name
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
```

This routine publishes the pair (port_name, service_name) so that an application may retrieve a system-supplied port_name using a well-known service_name.

The implementation must define the *scope* of a published service name, that is, the domain over which the service name is unique, and conversely, the domain over which the (port name, service name) pair may be retrieved. For instance, a service name may be unique to a job (where job is defined by a distributed operating system or batch scheduler), unique to a machine, or unique to a Kerberos realm. The scope may depend on the info argument to MPI_PUBLISH_NAME.

MPI permits publishing more than one service_name for a single port_name. On the other hand, if service_name has already been published within the scope determined by info, the behavior of MPI_PUBLISH_NAME is undefined. An MPI implementation may, through a mechanism in the info argument to MPI_PUBLISH_NAME, provide a way to allow multiple servers with the same service in the same scope. In this case, an implementation-defined policy will determine which of several port names is returned by MPI_LOOKUP_NAME.

Note that while service_name has a limited scope, determined by the implementation, port_name always has global scope within the communication universe used by the implementation (i.e., it is globally unique).

port_name should be the name of a port established by MPI_OPEN_PORT and not yet released by MPI_CLOSE_PORT. If it is not, the result is undefined.

*Advice to implementors.* In some cases, an MPI implementation may use a name service that a user can also access directly. In this case, a name published by MPI could easily conflict with a name published by a user. In order to avoid such conflicts, MPI implementations should mangle service names so that they are unlikely to conflict with user code that makes use of the same service. Such name mangling will of course be completely transparent to the user.

The following situation is problematic but unavoidable, if we want to allow implementations to use nameservers. Suppose there are multiple instances of "ocean" running on a machine. If the scope of a service name is confined to a job, then multiple oceans can coexist. If an implementation provides site-wide scope, however, multiple instances are not possible as all calls to MPI_PUBLISH_NAME after the first may fail. There is no universal solution to this.

To handle these situations, a high-quality implementation should make it possible to limit the domain over which names are published. (*End of advice to implementors.*)

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) |

**C binding**

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

**F08 binding**

```
MPI_Unpublish_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  service_name, port_name
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
```

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) |

**C binding**

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

**F08 binding**

```
MPI_Lookup_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  service_name
    TYPE(MPI_Info), INTENT(IN) ::  info
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) ::  port_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
```

This function retrieves a port_name published by MPI_PUBLISH_NAME with service_name. If service_name has not been published, it raises an error in the error class MPI_ERR_NAME. The application must supply a port_name buffer large enough to hold the largest possible port name (see discussion above under MPI_OPEN_PORT).

If an implementation allows multiple entries with the same service_name within the same scope, a particular port_name is chosen in a way determined by the implementation.

If the info argument was used with MPI_PUBLISH_NAME to tell the implementation how to publish names, a similar info argument may be required for MPI_LOOKUP_NAME.

### 10.4.5 Reserved Key Values

The following key values are reserved. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described.

ip_port Value contains IP port number at which to establish a port. (Reserved for MPI_OPEN_PORT only).

ip_address Value contains IP address at which to establish a port. If the address is not a valid IP address of the host on which the MPI_OPEN_PORT call is made, the results are undefined. (Reserved for MPI_OPEN_PORT only).

### 10.4.6 Client/Server Examples

#### Simplest Example — Completely Portable.

The following example shows the simplest way to use the client/server interface. It does not use service names at all.

On the server side:

```
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_size(MPI_COMM_WORLD, &size);
    if (size != 1) error(FATAL, "Server too big");
    MPI_Open_port(MPI_INFO_NULL, port_name);
    printf("server available at %s\n", port_name);
    while (1) {
        MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD,
                        &client);
        again = 1;
        while (again) {
            MPI_Recv(buf, MAX_DATA, MPI_DOUBLE,
                     MPI_ANY_SOURCE, MPI_ANY_TAG, client, &status);
            switch (status.MPI_TAG) {
                case 0: MPI_Comm_free(&client);
                        MPI_Close_port(port_name);
                        MPI_Finalize();
                        return 0;
                case 1: MPI_Comm_disconnect(&client);
                        again = 0;
                        break;
                case 2: /* do something */
                ...
                default:
                        /* Unexpected message type */
                        MPI_Abort(MPI_COMM_WORLD, 1);
                }
            }
        }
}
```

Here is the client.

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

```
int main(int argc, char **argv)
{
    MPI_Comm server;
    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 MPI_COMM_CONNECT routines, or spawn other MPI processes. MPI does not mandate this behavior, but strongly encourages it where technically feasible.

*Advice to implementors.* To start MPI processes belonging to the same MPI_COMM_WORLD requires some special coordination. The processes must be started at the "same" time, they must have a mechanism to establish communication, etc. Either the user or the operating system must take special steps beyond simply starting processes.

When an application enters MPI_INIT, clearly it must be able to determine if these special steps were taken. If a process enters MPI_INIT and determines that no special steps were taken (i.e., it has not been given the information to form an MPI_COMM_WORLD with other processes) it succeeds and forms a singleton MPI program, that is, one in which MPI_COMM_WORLD has size 1.

In some implementations, MPI may not be able to function without an "MPI environment." For example, MPI may require that daemons be running or MPI may not be able to work at all on the front-end of an MPP. In this case, an MPI implementation may either

1. Create the environment (e.g., start a daemon) or
2. Raise an error if it cannot create the environment and the environment has not been started independently.

A high-quality implementation will try to create a singleton MPI process and not raise an error.

(*End of advice to implementors.*)

### 10.5.3 MPI_APPNUM

There is a predefined attribute MPI_APPNUM of MPI_COMM_WORLD. In Fortran, the attribute is an integer value. In C, the attribute is a pointer to an integer value. If a process was spawned with MPI_COMM_SPAWN_MULTIPLE, MPI_APPNUM is the command number that generated the current process. Numbering starts from zero. If a process was spawned with MPI_COMM_SPAWN, it will have MPI_APPNUM equal to zero.

Additionally, if the process was not started by a spawn call, but by an implementation-specific startup mechanism that can handle multiple process specifications, MPI_APPNUM should be set to the number of the corresponding process specification. In particular, if it is started with

```
mpiexec spec0 [: spec1 : spec2 : ...]
```

MPI_APPNUM should be set to the number of the corresponding specification.

If an application was not spawned with MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, and MPI_APPNUM does not make sense in the context of the implementation-specific startup mechanism, MPI_APPNUM is not set.

MPI implementations may optionally provide a mechanism to override the value of MPI_APPNUM through the info argument. MPI reserves the following key for all SPAWN calls.

appnum Value contains an integer that overrides the default value for MPI_APPNUM in the child.

*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 do not share the same MPI_COMM_WORLD, may become disconnected (independent) if the communication path between them is broken by using MPI_COMM_DISCONNECT.

The following additional rules apply to MPI routines in other chapters:

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

MPI_COMM_DISCONNECT(comm)

| | | |
|---|---|---|
| INOUT | comm | communicator (handle) |

**C binding**

```
int MPI_Comm_disconnect(MPI_Comm *comm)
```

**F08 binding**

```
MPI_Comm_disconnect(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_COMM_DISCONNECT(COMM, IERROR)
    INTEGER COMM, IERROR
```

This function waits for all pending communication on comm to complete internally, deallocates the communicator object, and sets the handle to MPI_COMM_NULL. It is a collective operation.

It may not be called with the communicator MPI_COMM_WORLD or MPI_COMM_SELF.

MPI_COMM_DISCONNECT may be called only if all communication is complete and matched, so that buffered data can be delivered to its destination. This requirement is the same as for MPI_FINALIZE.

MPI_COMM_DISCONNECT has the same action as MPI_COMM_FREE, except that it waits for pending communication to finish internally and enables the guarantee about the behavior of disconnected processes.

*Advice to users.* To disconnect two processes you may need to call MPI_COMM_DISCONNECT, MPI_WIN_FREE, and MPI_FILE_CLOSE to remove all communication paths between the two processes. Note that it may be necessary to disconnect several communicators (or to free several windows or files) before two processes are completely independent. (*End of advice to users.*)

*Rationale.* It would be nice to be able to use MPI_COMM_FREE instead, but that function explicitly does not wait for pending communication to complete. (*End of rationale.*)

### 10.5.5 Another Way to Establish MPI Communication

MPI_COMM_JOIN(fd, intercomm)

| | | |
|---|---|---|
| IN | fd | socket file descriptor (integer) |
| OUT | intercomm | new intercommunicator (handle) |

**C binding**

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
```

**F08 binding**

```
MPI_Comm_join(fd, intercomm, ierror)
    INTEGER, INTENT(IN) :: fd
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

MPI_COMM_JOIN(fd, intercomm)

| | | |
|---|---|---|
| IN | fd | socket file descriptor |
| OUT | intercomm | new intercommunicator (handle) |

**C binding**

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
```

**F08 binding**

```
MPI_Comm_join(fd, intercomm, ierror)
    INTEGER, INTENT(IN) :: fd
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

MPI_COMM_JOIN is intended for MPI implementations that exist in an environment supporting the Berkeley Socket interface [45, 49]. 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.

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). Nonblocking 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 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 to update at other processes. However, the programmer may not be able to easily determine which data in a process may need to be accessed or to be updated by operations executed by a different process, and may not even know which processes may perform such updates. Thus, the transfer parameters are all available only on one side. Regular send/receive communication requires matching operations by sender and receiver. In order to issue the matching operations, an application needs to distribute the transfer parameters. This distribution may require all processes to participate in a time-consuming global computation, or to poll for potential communication requests to receive and upon which to act periodically. The use of RMA communication mechanisms avoids the need for global computations or explicit polling. A generic example of this nature is the execution of an assignment of the form `A = B(map)`, where `map` is a permutation vector, and `A`, `B`, and `map` are distributed in the same manner.

Message-passing communication achieves two effects: *communication* of data from sender to receiver and *synchronization* of sender with receiver. The RMA design separates these two functions. The following communication calls are provided:

- Remote write: MPI_PUT, MPI_RPUT
- Remote read: MPI_GET, MPI_RGET
- Remote update: MPI_ACCUMULATE, MPI_RACCUMULATE
- Remote read and update: MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP
- Remote atomic swap operations: MPI_COMPARE_AND_SWAP

This chapter refers to an operations set that includes all remote update, remote read and update, and remote atomic swap operations as "accumulate" operations.

MPI supports two fundamentally different *memory models*: *separate* and *unified*. The separate model makes no assumption about memory consistency and is highly portable. This model is similar to that of weakly coherent memory systems: the user must impose correct ordering of memory accesses through synchronization calls. The unified model can exploit cache-coherent hardware and hardware-accelerated, one-sided operations that are commonly available in high-performance systems. The two different models are discussed in detail in Section 11.4. Both models support several synchronization calls to support different synchronization styles.

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

We shall denote by **origin** the process that performs the call, and by **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

MPI provides the following window initialization functions: MPI_WIN_CREATE, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_CREATE_DYNAMIC, which are collective on an intracommunicator. MPI_WIN_CREATE allows each process to specify a "window" in its memory that is made accessible to accesses by remote processes. The call returns an opaque object that represents the group of processes that own and access the set of windows, and the attributes of each window, as specified by the initialization call. MPI_WIN_ALLOCATE differs from MPI_WIN_CREATE in that the user does not pass allocated memory; MPI_WIN_ALLOCATE returns a pointer to memory allocated by the MPI implementation. MPI_WIN_ALLOCATE_SHARED differs from MPI_WIN_ALLOCATE in that the allocated memory can be accessed from all processes in the window's group with direct load/store instructions. Some restrictions may apply to the specified communicator. MPI_WIN_CREATE_DYNAMIC creates a window that allows the user to dynamically control which memory is exposed by the window.

### 11.2.1 Window Creation

MPI_WIN_CREATE(base, size, disp_unit, info, comm, win)

| | | |
|---|---|---|
| IN | base | initial address of window (choice) |
| IN | size | size of window in bytes (non-negative integer) |
| IN | disp_unit | local unit size for displacements, in bytes (positive integer) |
| IN | info | info argument (handle) |
| IN | comm | intra-communicator (handle) |
| OUT | win | window object returned by the call (handle) |

**C binding**

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, MPI_Win *win)
```

**F08 binding**

```
MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    INTEGER, INTENT(IN) :: disp_unit
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Win), INTENT(OUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_CREATE(BASE, SIZE, DISP_UNIT, INFO, COMM, WIN, IERROR)
    <type> BASE(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
```

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. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see also Section 17.1.12). 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, rather than a basic integer type, to allow windows that span more memory than can be described with a basic integer type. (*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 latter 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 keys are predefined:

no_locks — if set to true, then the implementation may assume that passive target synchronization (i.e., MPI_WIN_LOCK, MPI_WIN_LOCK_ALL) will not be used on the given window. 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.

accumulate_ordering — controls the ordering of accumulate operations at the target. See Section 11.7.2 for details.

accumulate_ops — if set to same_op, the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation. If set to same_op_no_op, then the implementation will assume that all concurrent accumulate calls to the same target address will use the same operation or MPI_NO_OP. This can eliminate the need to protect access for certain operation types where the hardware can guarantee atomicity. The default is same_op_no_op.

same_size — if set to true, then the implementation may assume that the argument size is identical on all processes, and that all processes have provided this info key with the same value.

same_disp_unit — if set to true, then the implementation may assume that the argument disp_unit is identical on all processes, and that all processes have provided this info key with the same value.

*Advice to users.* The info query mechanism described in Section 11.2.7 can be used to query the specified info arguments for windows that have been passed to a library. It is recommended that libraries check attached info keys for each passed window. (*End of advice to users.*)

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 undefined results.

*Rationale.* The reason for specifying the memory that may be accessed from another process in an RMA operation is to permit the programmer to specify what memory can be a target of RMA operations and for the implementation to enforce that specification. For example, with this definition, a server process can safely allow a client process to use RMA operations, knowing that (under the assumption that the MPI implementation does enforce the specified limits on the exposed memory) an error in the client cannot affect any memory other than what was explicitly exposed. (*End of rationale.*)

*Advice to users.* A window can be created in any part of the process memory. However, on some systems, the performance of windows in memory allocated by MPI_ALLOC_MEM (Section 8.2) will be better. Also, on some systems, performance is improved when window boundaries are aligned at “natural” boundaries (word, double-word, cache line, page frame, etc.). (*End of advice to users.*)

*Advice to implementors.* In cases where RMA operations use different mechanisms in different memory areas (e.g., load/store in a shared memory segment, and an asynchronous handler in private memory), the MPI_WIN_CREATE call needs to figure out which type of memory is used for the window. To do so, MPI maintains, internally, the list of memory segments allocated by MPI_ALLOC_MEM, or by other, implementation-specific, mechanisms, together with information on the type of memory segment allocated. When a call to MPI_WIN_CREATE occurs, then MPI checks which segment contains each window, and decides, accordingly, which mechanism to use for RMA operations.

Vendors may provide additional, implementation-specific mechanisms to allocate or to specify memory regions that are preferable for use in one-sided communication. In particular, such mechanisms can be used to place static variables into such preferred regions.

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

### 11.2.2 Window That Allocates Memory

MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)

| | | |
|---|---|---|
| IN | size | size of window in bytes (non-negative integer) |
| IN | disp_unit | local unit size for displacements, in bytes (positive integer) |
| IN | info | info argument (handle) |
| IN | comm | intra-communicator (handle) |
| OUT | baseptr | initial address of window (choice) |
| OUT | win | window object returned by the call (handle) |

**C binding**

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, void *baseptr, MPI_Win *win)
```

**F08 binding**

```
MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    INTEGER, INTENT(IN) :: disp_unit
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    TYPE(C_PTR), INTENT(OUT) ::  baseptr
    TYPE(MPI_Win), INTENT(OUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

This is a collective call executed by all processes in the group of comm. On each process, it allocates memory of at least size bytes, returns a pointer to it, and returns a window object that can be used by all processes in comm to perform RMA operations. The returned memory consists of size bytes local to each process, starting at address baseptr and is associated with the window as if the user called MPI_WIN_CREATE on existing memory. The size argument may be different at each process and size = 0 is valid; however, a library might allocate and expose more memory in order to create a fast, globally symmetric allocation. The discussion of and rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section 8.2 also apply to MPI_WIN_ALLOCATE; in particular, see the rationale in Section 8.2 for an explanation of the type used for baseptr.

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, but with a different specific procedure name:

```
INTERFACE MPI_WIN_ALLOCATE
    SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
                                WIN, IERROR)
        IMPORT ::  MPI_ADDRESS_KIND
        INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
                                     WIN, IERROR)
        USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
        IMPORT ::  MPI_ADDRESS_KIND
        INTEGER ::  DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE
        TYPE(C_PTR) ::  BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_CPTR. The implied specific procedure names are described in Section 17.1.5.

*Rationale.* By allocating (potentially aligned) memory instead of allowing the user to pass in an arbitrary buffer, this call can improve the performance for systems with remote direct memory access. This also permits the collective allocation of memory and supports what is sometimes called the "symmetric allocation" model that can be more scalable (for example, the implementation can arrange to return an address for the allocated memory that is the same on all processes). (*End of rationale.*)

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE and MPI_ALLOC_MEM.

### 11.2.3 Window That Allocates Shared Memory

MPI_WIN_ALLOCATE_SHARED(size, disp_unit, info, comm, baseptr, win)

| | | |
|---|---|---|
| IN | size | size of local window in bytes (non-negative integer) |
| IN | disp_unit | local unit size for displacements, in bytes (positive integer) |
| IN | info | info argument (handle) |
| IN | comm | intra-communicator (handle) |
| OUT | baseptr | address of local allocated window segment (choice) |
| OUT | win | window object returned by the call (handle) |

**C binding**

```
int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, void *baseptr, MPI_Win *win)
```

**F08 binding**

```
MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    INTEGER, INTENT(IN) :: disp_unit
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    TYPE(MPI_Win), INTENT(OUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

This is a collective call executed by all processes in the group of comm. On each process, it allocates memory of at least size bytes that is shared among all processes in comm, and returns a pointer to the locally allocated segment in baseptr that can be used for load/store accesses on the calling process. The locally allocated memory can be the target of load/store accesses by remote processes; the base pointers for other processes can be queried using the function MPI_WIN_SHARED_QUERY. The call also returns a window object that can be used by all processes in comm to perform RMA operations. The size argument may be different at each process and size = 0 is valid. It is the user's responsibility to ensure that the communicator comm represents a group of processes that can create a shared memory segment that can be accessed by all processes in the group. The discussions of rationales for MPI_ALLOC_MEM and MPI_FREE_MEM in Section 8.2 also apply to MPI_WIN_ALLOCATE_SHARED; in particular, see the rationale in Section 8.2

for an explanation of the type used for baseptr. The allocated memory is contiguous across process ranks unless the info key alloc_shared_noncontig is specified. Contiguous across process ranks means that the first address in the memory segment of process $i$ is consecutive with the last address in the memory segment of process $i-1$. This may enable the user to calculate remote address offsets with local information only.

If the Fortran compiler provides `TYPE(C_PTR)`, then the following generic interface must be provided in the `mpi` module and should be provided in `mpif.h` through overloading, i.e., with the same routine name as the routine with `INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR`, but with a different specific procedure name:

```
INTERFACE MPI_WIN_ALLOCATE_SHARED
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
                                       BASEPTR, WIN, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
                                            BASEPTR, WIN, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_WIN_ALLOCATE_SHARED_CPTR. The implied specific procedure names are described in Section 17.1.5.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE, MPI_WIN_ALLOCATE, and MPI_ALLOC_MEM. The additional info key alloc_shared_noncontig allows the library to optimize the layout of the shared memory segments in memory.

> *Advice to users.* If the info key alloc_shared_noncontig is not set to true, the allocation strategy is to allocate contiguous memory across process ranks. This may limit the performance on some architectures because it does not allow the implementation to modify the data layout (e.g., padding to reduce access latency). (*End of advice to users.*)

> *Advice to implementors.* If the user sets the info key alloc_shared_noncontig to true, the implementation can allocate the memory requested by each process in a location that is close to this process. This can be achieved by padding or allocating memory in special memory segments. Both techniques may make the address space across consecutive ranks noncontiguous. (*End of advice to implementors.*)

The consistency of load/store accesses from/to the shared memory as observed by the user program depends on the architecture. A consistent view can be created in the *unified* *memory model* (see Section 11.4) by utilizing the window synchronization functions (see Section 11.5) or explicitly completing outstanding store accesses (e.g., by calling MPI_WIN_FLUSH). MPI does not define semantics for accessing shared memory windows in the *separate memory model*.

MPI_WIN_SHARED_QUERY(win, rank, size, disp_unit, baseptr)

| | | |
|---|---|---|
| IN | win | shared memory window object (handle) |
| IN | rank | rank in the group of window win (non-negative integer) or MPI_PROC_NULL |
| OUT | size | size of the window segment (non-negative integer) |
| OUT | disp_unit | local unit size for displacements, in bytes (positive integer) |
| OUT | baseptr | address for load/store access to window segment (choice) |

**C binding**

```
int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
             int *disp_unit, void *baseptr)
```

**F08 binding**

```
MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)
    USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, INTENT(IN) ::  rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  size
    INTEGER, INTENT(OUT) ::  disp_unit
    TYPE(C_PTR), INTENT(OUT) ::  baseptr
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
    INTEGER WIN, RANK, DISP_UNIT, IERROR
    INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

This function queries the process-local address for remote memory segments created with MPI_WIN_ALLOCATE_SHARED. This function can return different process-local addresses for the same physical memory on different processes. The returned memory can be used for load/store accesses subject to the constraints defined in Section 11.7. This function can only be called with windows of flavor MPI_WIN_FLAVOR_SHARED. If the passed window is not of flavor MPI_WIN_FLAVOR_SHARED, the error MPI_ERR_RMA_FLAVOR is raised. When rank is MPI_PROC_NULL, the pointer, disp_unit, and size returned are the pointer, disp_unit, and size of the memory segment belonging the lowest rank that specified size $> 0$. If all processes in the group attached to the window specified size $= 0$, then the call returns size $= 0$ and a baseptr as if MPI_ALLOC_MEM was called with size $= 0$.

If the Fortran compiler provides TYPE(C_PTR), then the following generic interface must be provided in the mpi module and should be provided in mpif.h through overloading, i.e., with the same routine name as the routine with INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR, but with a different specific procedure name:

```
INTERFACE MPI_WIN_SHARED_QUERY
    SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
                                    BASEPTR, IERROR)
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER WIN, RANK, DISP_UNIT, IERROR
        INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
                                         BASEPTR, IERROR)
        USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
        IMPORT :: MPI_ADDRESS_KIND
        INTEGER :: WIN, RANK, DISP_UNIT, IERROR
        INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
        TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
END INTERFACE
```

The base procedure name of this overloaded function is MPI_WIN_SHARED_QUERY_CPTR. The implied specific procedure names are described in Section 17.1.5.

### 11.2.4 Window of Dynamically Attached Memory

The MPI-2 RMA model requires the user to identify the local memory that may be a target of RMA calls at the time the window is created. This has advantages for both the programmer (only this memory can be updated by one-sided operations and provides greater safety) and the MPI implementation (special steps may be taken to make one-sided access to such memory more efficient). However, consider implementing a modifiable linked list using RMA operations; as new items are added to the list, memory must be allocated. In a C or C++ program, this memory is typically allocated using `malloc` or `new` respectively. In MPI-2 RMA, the programmer must create a window with a predefined amount of memory and then implement routines for allocating memory from within the window's memory. In addition, there is no easy way to handle the situation where the predefined amount of memory turns out to be inadequate. To support this model, the routine MPI_WIN_CREATE_DYNAMIC creates a window that makes it possible to expose memory without remote synchronization. It must be used in combination with the local routines MPI_WIN_ATTACH and MPI_WIN_DETACH.

MPI_WIN_CREATE_DYNAMIC(info, comm, win)

| | | |
|---|---|---|
| IN | info | info argument (handle) |
| IN | comm | intra-communicator (handle) |
| OUT | win | window object returned by the call (handle) |

**C binding**

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

**F08 binding**

```
MPI_Win_create_dynamic(info, comm, win, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Win), INTENT(OUT) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
    INTEGER INFO, COMM, WIN, IERROR
```

This is a collective call executed by all processes in the group of comm. It returns a window win without memory attached. Existing process memory can be attached as described below. This routine returns a window object that can be used by these processes to perform RMA operations on attached memory. Because this window has special properties, it will sometimes be referred to as a *dynamic* window.

The info argument can be used to specify hints similar to the info argument for MPI_WIN_CREATE.

In the case of a window created with MPI_WIN_CREATE_DYNAMIC, the target_disp for all RMA functions is the address at the target; i.e., the effective window_base is MPI_BOTTOM and the disp_unit is one. For dynamic windows, the target_disp argument to RMA communication operations is not restricted to non-negative values. Users should use MPI_GET_ADDRESS at the target process to determine the address of a target memory location and communicate this address to the origin process.

*Advice to users.* Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (*End of advice to users.*)

*Advice to implementors.* In environments with heterogeneous data representations, care must be exercised in communicating addresses between processes. For example, it is possible that an address valid at the target process (for example, a 64-bit pointer) cannot be expressed as an address at the origin (for example, the origin uses 32-bit pointers). For this reason, a portable MPI implementation should ensure that the type MPI_AINT (see Table 3.3) is able to store addresses from any process. (*End of advice to implementors.*)

Memory at the target cannot be accessed with this window until that memory has been attached using the function MPI_WIN_ATTACH. That is, in addition to using MPI_WIN_CREATE_DYNAMIC to create an MPI window, the user must use MPI_WIN_ATTACH before any local memory may be the target of an MPI RMA operation. Only memory that is currently accessible may be attached.

MPI_WIN_ATTACH(win, base, size)

| | | |
|---|---|---|
| IN | win | window object (handle) |
| IN | base | initial address of memory to be attached |
| IN | size | size of memory to be attached in bytes |

**C binding**

```
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
```

**F08 binding**

```
MPI_Win_attach(win, base, size, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
    INTEGER WIN, IERROR
    <type> BASE(*)
    INTEGER (KIND=MPI_ADDRESS_KIND) SIZE
```

Attaches a local memory region beginning at base for remote access within the given window. The memory region specified must not contain any part that is already attached to the window win, that is, attaching overlapping memory concurrently within the same window is erroneous. The argument win must be a window that was created with MPI_WIN_CREATE_DYNAMIC. The local memory region attached to the window consists of size bytes, starting at address base. In C, base is the starting address of a memory region. In Fortran, one can pass the first element of a memory region or a whole array, which must be 'simply contiguous' (for 'simply contiguous,' see Section 17.1.12). Multiple (but non-overlapping) memory regions may be attached to the same window.

*Rationale.* Requiring that memory be explicitly attached before it is exposed to one-sided access by other processes can simplify implementations and improve performance. The ability to make memory available for RMA operations without requiring a collective MPI_WIN_CREATE call is needed for some one-sided programming models. (*End of rationale.*)

*Advice to users.* Attaching memory to a window may require the use of scarce resources; thus, attaching large regions of memory is not recommended in portable programs. Attaching memory to a window may fail if sufficient resources are not available; this is similar to the behavior of MPI_ALLOC_MEM.

The user is also responsible for ensuring that MPI_WIN_ATTACH at the target has returned before a process attempts to target that memory with an MPI RMA call.

Performing an RMA operation to memory that has not been attached to a window created with MPI_WIN_CREATE_DYNAMIC is erroneous. (*End of advice to users.*)

*Advice to implementors.* A high-quality implementation will attempt to make as much memory available for attaching as possible. Any limitations should be documented by the implementor. (*End of advice to implementors.*)

Attaching memory is a local operation as defined by MPI, which means that the call is not collective and completes without requiring any MPI routine to be called in any other process. Memory may be detached with the routine MPI_WIN_DETACH. After memory has been detached, it may not be the target of an MPI RMA operation on that window (unless the memory is re-attached with MPI_WIN_ATTACH).

MPI_WIN_DETACH(win, base)

| | | |
|---|---|---|
| IN | win | window object (handle) |
| IN | base | initial address of memory to be detached |

**C binding**

```
int MPI_Win_detach(MPI_Win win, const void *base)
```

**F08 binding**

```
MPI_Win_detach(win, base, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: base
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_DETACH(WIN, BASE, IERROR)
    INTEGER WIN, IERROR
    <type> BASE(*)
```

Detaches a previously attached memory region beginning at base. The arguments base and win must match the arguments passed to a previous call to MPI_WIN_ATTACH.

*Advice to users.* Detaching memory may permit the implementation to make more efficient use of special memory or provide memory that may be needed by a subsequent MPI_WIN_ATTACH. Users are encouraged to detach memory that is no longer needed. Memory should be detached before it is freed by the user. (*End of advice to users.*)

Memory becomes detached when the associated dynamic memory window is freed, see Section 11.2.5.

### 11.2.5 Window Destruction

MPI_WIN_FREE(win)

| | | |
|---|---|---|
| INOUT | win | window object (handle) |

**C binding**

```
int MPI_Win_free(MPI_Win *win)
```

**F08 binding**

```
MPI_Win_free(win, ierror)
    TYPE(MPI_Win), INTENT(INOUT) :: win
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR
```

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: e.g., 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. The memory associated with windows created by a call to MPI_WIN_CREATE may be freed after the call returns. If the window was created with MPI_WIN_ALLOCATE, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_ALLOCATE. If the window was created with MPI_WIN_ALLOCATE_SHARED, MPI_WIN_FREE will free the window memory that was allocated in MPI_WIN_ALLOCATE_SHARED.

Freeing a window that was created with a call to MPI_WIN_CREATE_DYNAMIC detaches all associated memory; i.e., it has the same effect as if all attached memory was detached by calls to MPI_WIN_DETACH.

> *Advice to implementors.* MPI_WIN_FREE requires a barrier synchronization: no process can return from free until all processes in the group of win call free. This ensures that no process will attempt to access a remote window (e.g., with lock/unlock) after it was freed. The only exception to this rule is when the user sets the no_locks info key to true when creating the window. In that case, an MPI implementation may free the local window without barrier synchronization. (*End of advice to implementors.*)

### 11.2.6 Window Attributes

The following 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. |
| MPI_WIN_CREATE_FLAVOR | how the window was created. |
| MPI_WIN_MODEL | memory model for 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), MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag), MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag), and MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag) will return in base a pointer to the start of the window win, and will return in size, disp_unit, create_kind, and memory_model pointers to the size, displacement unit of the window, the kind of routine used to create the window, and the memory model, respectively. A detailed listing of the type of the pointer in the attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR is shown in Table 11.1.

| Attribute | C Type |
|---|---|
| MPI_WIN_BASE | void * |
| MPI_WIN_SIZE | MPI_Aint * |
| MPI_WIN_DISP_UNIT | int * |
| MPI_WIN_CREATE_FLAVOR | int * |
| MPI_WIN_MODEL | int * |

Table 11.1: C types of attribute value argument to MPI_WIN_GET_ATTR and MPI_WIN_SET_ATTR.

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), MPI_WIN_GET_ATTR(win, MPI_WIN_DISP_UNIT, disp_unit, flag, ierror), MPI_WIN_GET_ATTR(win, MPI_WIN_CREATE_FLAVOR, create_kind, flag, ierror), and MPI_WIN_GET_ATTR(win, MPI_WIN_MODEL, memory_model, flag, ierror) will return in base, size, disp_unit, create_kind, and memory_model the (integer representation of) the base address, the size, the displacement unit of the window win, the kind of routine used to create the window, and the memory model, respectively.

The values of create_kind are

| | |
|---|---|
| MPI_WIN_FLAVOR_CREATE | Window was created with MPI_WIN_CREATE. |
| MPI_WIN_FLAVOR_ALLOCATE | Window was created with MPI_WIN_ALLOCATE. |
| MPI_WIN_FLAVOR_DYNAMIC | Window was created with MPI_WIN_CREATE_DYNAMIC. |
| MPI_WIN_FLAVOR_SHARED | Window was created with MPI_WIN_ALLOCATE_SHARED. |

The values of memory_model are MPI_WIN_SEPARATE and MPI_WIN_UNIFIED. The meaning of these is described in Section 11.4.

In the case of windows created with MPI_WIN_CREATE_DYNAMIC, the base address is MPI_BOTTOM and the size is 0. In C, pointers are returned, and in Fortran, the values are returned, for the respective attributes. (The window attribute access functions are defined in Section 6.7.3.) The value returned for an attribute on a window is constant over the lifetime of the window.

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

MPI_WIN_GET_GROUP(win, group)

| | | |
|---|---|---|
| IN | win | window object (handle) |
| OUT | group | group of processes which share access to the window (handle) |

**C binding**

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

**F08 binding**

```
MPI_Win_get_group(win, group, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
    INTEGER WIN, GROUP, IERROR
```

MPI_WIN_GET_GROUP returns a duplicate of the group of the communicator used to create the window associated with win. The group is returned in group.

### 11.2.7 Window Info

Hints specified via info (see Section 9) allow a user to provide information to direct optimization. Providing hints may enable an implementation to deliver increased performance or use system resources more efficiently. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_WIN_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per window basis, in window creation functions and MPI_WIN_SET_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_WIN_SET_INFO there will be no effect on previously set or default 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 the hint. (*End of advice to implementors.*)

MPI_WIN_SET_INFO(win, info)

| | | |
|---|---|---|
| INOUT | win | window object (handle) |
| IN | info | info object (handle) |

**C binding**

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
```

**F08 binding**

```
MPI_Win_set_info(win, info, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_SET_INFO(WIN, INFO, IERROR)
    INTEGER WIN, INFO, IERROR
```

MPI_WIN_SET_INFO updates the hints of the window associated with win using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_WIN_SET_INFO. The call is collective on the group of win. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

*Advice to users.* Some info items that an implementation can use when it creates a window cannot easily be changed once the window has been created. Thus, an implementation may ignore hints issued in this call that it would have accepted in a creation call. An implementation may also be unable to update certain info hints in a call to MPI_WIN_SET_INFO. MPI_WIN_GET_INFO can be used to determine whether info changes were ignored by the implementation. (*End of advice to users.*)

MPI_WIN_GET_INFO(win, info_used)

| | | |
|---|---|---|
| IN | win | window object (handle) |
| OUT | info_used | new info object (handle) |

**C binding**

```
int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
```

**F08 binding**

```
MPI_Win_get_info(win, info_used, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Info), INTENT(OUT) ::  info_used
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
    INTEGER WIN, INFO_USED, IERROR
```

MPI_WIN_GET_INFO returns a new info object containing the hints of the window associated with win. The current setting of all hints related to this window is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pair. The user is responsible for freeing info_used via MPI_INFO_FREE.

## 11.3 Communication Calls

MPI supports the following RMA communication calls: MPI_PUT and MPI_RPUT transfer data from the caller memory (origin) to the target memory; MPI_GET and MPI_RGET transfer data from the target memory to the caller memory; MPI_ACCUMULATE and MPI_RACCUMULATE update locations in the target memory, e.g., by adding to these locations values sent from the caller memory; MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP perform atomic read-modify-write and return the data before the accumulate operation; and MPI_COMPARE_AND_SWAP performs a remote atomic compare and swap operation. These operations are *nonblocking*: the call initiates the transfer, but the transfer may continue after the call returns. The transfer is completed, at the origin or both the origin and the target, when a subsequent *synchronization* call is issued by the caller on the involved window object. These synchronization calls are described in Section 11.5. Transfers can also be completed with calls to flush routines; see Section 11.5.4 for details. For the MPI_RPUT, MPI_RGET, MPI_RACCUMULATE, and MPI_RGET_ACCUMULATE calls, the transfer can be locally completed by using the MPI test or wait operations described in Section 3.7.3.

The local communication buffer of an RMA call should not be updated, and the local communication buffer of a get call should not be accessed after the RMA call until the operation completes at the origin.

The resulting data values, or outcome, of concurrent conflicting accesses to the same memory locations is undefined; if a location is updated by a put or accumulate operation, then the outcome of loads or other RMA operations is undefined 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, the outcome of concurrent load/store and RMA updates to the same memory location is undefined. These restrictions are described in more detail in Section 11.7.

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 RMA 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 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 all MPI RMA communication calls. The effect is the same as for MPI_PROC_NULL in MPI point-to-point communication. After any RMA operation with rank MPI_PROC_NULL, it is still necessary to finish the RMA epoch with the synchronization method that started the epoch.

### 11.3.1 Put

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

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

| | | |
|---|---|---|
| IN | origin_addr | initial address of origin buffer (choice) |
| IN | origin_count | number of entries in origin buffer (non-negative integer) |
| IN | origin_datatype | datatype of each entry in origin buffer (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | win | window object used for communication (handle) |

**C binding**

```
int MPI_Put(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win)
```

**F08 binding**

```
MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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
```

Transfers origin_count successive entries of the type specified by the origin_datatype, starting at address origin_addr on the origin node, to the target node specified by the win, target_rank pair. The data are written in the target buffer at address target_addr = window_base + target_disp × disp_unit, where window_base and disp_unit are the base address and window displacement unit specified at window initialization, by the target process.

The target buffer is specified by the arguments target_count and target_datatype.

The data transfer is the same as that which would occur if the origin process executed a send operation with arguments origin_addr, origin_count, origin_datatype, target_rank, tag, comm, and the target process executed a receive operation with arguments target_addr, target_count, target_datatype, source, tag, comm, where target_addr is the target buffer address computed as explained above, the values of tag are arbitrary valid matching tag values, and comm is a communicator for the group of win.

The communication must satisfy the same constraints as for a similar message-passing communication. The 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 or in attached memory in a dynamic window.

The 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 operations.

*Advice to users.* The target_datatype argument is a handle to a datatype object that is defined at the origin process, even though it defines a data layout in the target process memory. This causes no problems in a homogeneous environment, or in a heterogeneous environment if only portable datatypes are used (portable datatypes are defined in Section 2.4).

The performance of a put transfer can be significantly affected, on some systems, by 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 or MPI_WIN_ALLOCATE may be much faster on shared memory systems; transfers from contiguous buffers will be faster on most, if not all, systems; the alignment of the communication buffers may also impact performance. (*End of advice to users.*)

*Advice to implementors.* A high-quality implementation will attempt to prevent remote accesses to memory outside the window that was exposed by the process. This is important both for debugging purposes and for protection with client-server codes that use RMA. That is, a high-quality implementation will check, if possible, window bounds on each RMA call, and raise an MPI exception at the origin call if an out-of-bound situation occurs. Note that the condition can be checked at the origin. Of course, the added safety achieved by such checks has to be weighed against the added cost of such checks. (*End of advice to implementors.*)

### 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 (non-negative integer) |
| IN | origin_datatype | datatype of each entry in origin buffer (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from window start to the beginning of the target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | win | window object used for communication (handle) |

**C binding**

```
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)
```

**F08 binding**

```
MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
            target_disp, target_count, target_datatype, win, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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
```

Similar to MPI_PUT, except that the direction of data transfer is reversed. Data are copied from the target memory to the origin. The origin_datatype may not specify overlapping entries in the origin buffer. The target buffer must be contained within the target window or within attached memory in a dynamic window, and the copied data must fit, without truncation, in the origin buffer.

### 11.3.3 Examples for Communication Calls

These examples show the use of the MPI_GET function. As all MPI RMA communication functions are nonblocking, they must be completed. In the following, this is accomplished with the routine MPI_WIN_FENCE, introduced in Section 11.5.

**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
     ttype(p), tindex(m),      & ! used to construct target datatypes
     count(p), total(p),       &
     disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint

! This part does the work that depends on the locations of B.
! Can be reused while this does not change

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,   &
                    comm, win, ierr)

! This part does the work that depends on the value of map and
! the locations of the arrays.
! Can be reused while these do not change

! Compute number of entries to be received from each process

DO i=1,p
  count(i) = 0
END DO
DO i=1,m
  j = map(i)/m+1
  count(j) = count(j)+1
END DO

total(1) = 0
DO i=2,p
  total(i) = total(i-1) + count(i-1)
END DO
```

```
DO i=1,p
  count(i) = 0
END DO

! compute origin and target indices of entries.
! entry i at current process is received from location
! k at process (j-1), where map(i) = (j-1)*m + (k-1),
! j = 1..p and k = 1..m

DO i=1,m
  j = map(i)/m+1
  k = MOD(map(i),m)+1
  count(j) = count(j)+1
  oindex(total(j) + count(j)) = i
  tindex(total(j) + count(j)) = k
END DO

! create origin and target datatypes for each get operation
DO i=1,p
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                   oindex(total(i)+1:total(i)+count(i)), &
                                   MPI_REAL, otype(i), ierr)
  CALL MPI_TYPE_COMMIT(otype(i), ierr)
  CALL MPI_TYPE_CREATE_INDEXED_BLOCK(count(i), 1, &
                                   tindex(total(i)+1:total(i)+count(i)), &
                                   MPI_REAL, ttype(i), ierr)
  CALL MPI_TYPE_COMMIT(ttype(i), ierr)
END DO

! this part does the assignment itself
CALL MPI_WIN_FENCE(0, win, ierr)
disp_aint = 0
DO i=1,p
  CALL MPI_GET(A, 1, otype(i), i-1, disp_aint, 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 disp_int, win, ierr
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
disp_int = realextent
size = m * realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  disp_aint = MOD(map(i),m)
  CALL MPI_GET(A(i), 1, MPI_REAL, j, disp_aint, 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 it. This will allow, for example, the accumulation of a sum by having all involved processes add their contributions to the sum variable in the memory of one process. The accumulate functions have slightly different semantics with respect to overlapping data accesses than the put and get functions; see Section 11.7 for details.

Accumulate Function

MPI_ACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win)

| | | |
|---|---|---|
| IN | origin_addr | initial address of buffer (choice) |
| IN | origin_count | number of entries in buffer (non-negative integer) |
| IN | origin_datatype | datatype of each entry (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to beginning of target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | op | reduce operation (handle) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

**F08 binding**

```
MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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
```

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. The parameter 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 can be used only in MPI_ACCUMULATE, MPI_RACCUMULATE, MPI_GET_ACCUMULATE, MPI_FETCH_AND_OP, and MPI_RGET_ACCUMULATE, but not in collective reduction operations such as MPI_REDUCE.

> *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 $\mathtt{B(j)} = \sum_{\mathtt{map(i)=j}} \mathtt{A(i)}$. The arrays A, B, and map are distributed in the same manner. We write the simple version.

```
SUBROUTINE SUM(A, B, map, m, comm, p)
USE MPI
INTEGER m, map(m), comm, p, win, ierr, disp_int
REAL A(m), B(m)
INTEGER (KIND=MPI_ADDRESS_KIND) lowerbound, size, realextent, disp_aint

CALL MPI_TYPE_GET_EXTENT(MPI_REAL, lowerbound, realextent, ierr)
size = m * realextent
disp_int = realextent
CALL MPI_WIN_CREATE(B, size, disp_int, MPI_INFO_NULL,  &
                    comm, win, ierr)

CALL MPI_WIN_FENCE(0, win, ierr)
DO i=1,m
  j = map(i)/m
  disp_aint = MOD(map(i),m)
  CALL MPI_ACCUMULATE(A(i), 1, MPI_REAL, j, disp_aint, 1, MPI_REAL,   &
                      MPI_SUM, win, ierr)
END DO
CALL MPI_WIN_FENCE(0, win, ierr)

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

This code is identical to the code in Example 11.2, except that a call to get has been replaced by a call to accumulate. (Note that, if `map` is one-to-one, the code computes `B` = `A(map`$^{-1}$`)`, which is the reverse assignment to the one computed in that previous example.) In a similar manner, we can replace in Example 11.1, the call to get by a call to accumulate, thus performing the computation with only one communication between any two processes.

### Get Accumulate Function

It is often useful to have fetch-and-accumulate semantics such that the remote data is returned to the caller before the sent data is accumulated into the remote data. The get and accumulate steps are executed atomically for each basic element in the datatype (see Section 11.7 for details). The predefined operation MPI_REPLACE provides fetch-and-set behavior.

MPI_GET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr, result_count, result_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 origin buffer (non-negative integer) |
| IN | origin_datatype | datatype of each entry in origin buffer (handle) |
| OUT | result_addr | initial address of result buffer (choice) |
| IN | result_count | number of entries in result buffer (non-negative integer) |
| IN | result_datatype | datatype of each entry in result buffer (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to beginning of target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | op | reduce operation (handle) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Get_accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, void *result_addr,
             int result_count, MPI_Datatype result_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

**F08 binding**

```
MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
             result_count, result_datatype, target_rank, target_disp,
             target_count, target_datatype, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    INTEGER, INTENT(IN) ::  origin_count, result_count, target_rank,
              target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype,
              result_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
             RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR
```

Accumulate origin_count elements of type origin_datatype from the origin buffer ( origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation, specified by target_disp, target_count, and target_datatype. The data transferred from origin to target must fit, without truncation, in the target buffer. Likewise, the data copied from target to origin must fit, without truncation, in the result buffer.

The origin and result buffers (origin_addr and result_addr) must be disjoint. Each datatype argument must be a predefined datatype or a derived datatype where all basic components are of the same predefined datatype. All datatype arguments must be constructed from the same predefined datatype. The operation op applies to elements of that predefined type. target_datatype must not specify overlapping entries, and the target buffer must fit in the target window or in attached memory in a dynamic window. The operation is executed atomically for each basic datatype; see Section 11.7 for details.

Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE can be specified as op. User-defined functions cannot be used. A new predefined operation, MPI_NO_OP, is defined. It corresponds to the associative function $f(a, b) = a$; i.e., the current value in the target memory is returned in the result buffer at the origin and no operation is performed on the target buffer. When MPI_NO_OP is specified as the operation, the origin_addr, origin_count, and origin_datatype arguments are ignored. MPI_NO_OP can be used only in MPI_GET_ACCUMULATE, MPI_RGET_ACCUMULATE, and MPI_FETCH_AND_OP. MPI_NO_OP cannot be used in MPI_ACCUMULATE, MPI_RACCUMULATE, or collective reduction operations, such as MPI_REDUCE and others.

*Advice to users.* MPI_GET is similar to MPI_GET_ACCUMULATE, with the operation MPI_NO_OP. Note, however, that MPI_GET and MPI_GET_ACCUMULATE have different constraints on concurrent updates. (*End of advice to users.*)

### Fetch and Op Function

The generic functionality of MPI_GET_ACCUMULATE might limit the performance of fetch-and-increment or fetch-and-add calls that might be supported by special hardware operations. MPI_FETCH_AND_OP thus allows for a fast implementation of a commonly used subset of the functionality of MPI_GET_ACCUMULATE.

MPI_FETCH_AND_OP(origin_addr, result_addr, datatype, target_rank, target_disp, op, win)

| | | |
|---|---|---|
| IN | origin_addr | initial address of buffer (choice) |
| OUT | result_addr | initial address of result buffer (choice) |
| IN | datatype | datatype of the entry in origin, result, and target buffers (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to beginning of target buffer (non-negative integer) |
| IN | op | reduce operation (handle) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
             MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
             MPI_Op op, MPI_Win win)
```

**F08 binding**

```
MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
             target_disp, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  target_rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
             TARGET_DISP, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER DATATYPE, TARGET_RANK, OP, WIN, IERROR
```

Accumulate one element of type datatype from the origin buffer (origin_addr) to the buffer at offset target_disp, in the target window specified by target_rank and win, using the operation op and return in the result buffer result_addr the content of the target buffer before the accumulation.

The origin and result buffers (origin_addr and result_addr) must be disjoint. Any of the predefined operations for MPI_REDUCE, as well as MPI_NO_OP or MPI_REPLACE, can be specified as op; user-defined functions cannot be used. The datatype argument must be a predefined datatype. The operation is executed atomically.

### Compare and Swap Function

Another useful operation is an atomic compare and swap where the value at the origin is compared to the value at the target, which is atomically replaced by a third value only if the values at origin and target are equal.

MPI_COMPARE_AND_SWAP(origin_addr, compare_addr, result_addr, datatype, target_rank, target_disp, win)

| | | |
|---|---|---|
| IN | origin_addr | initial address of buffer (choice) |
| IN | compare_addr | initial address of compare buffer (choice) |
| OUT | result_addr | initial address of result buffer (choice) |
| IN | datatype | datatype of the element in all buffers (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to beginning of target buffer (non-negative integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
             void *result_addr, MPI_Datatype datatype, int target_rank,
             MPI_Aint target_disp, MPI_Win win)
```

**F08 binding**

```
MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
             target_rank, target_disp, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  compare_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  target_rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
             TARGET_RANK, TARGET_DISP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
```

```
INTEGER DATATYPE, TARGET_RANK, WIN, IERROR
```

This function compares one element of type datatype in the compare buffer compare_addr with the buffer at offset target_disp in the target window specified by target_rank and win and replaces the value at the target with the value in the origin buffer origin_addr if the compare buffer and the target buffer are identical. The original value at the target is returned in the buffer result_addr. The parameter datatype must belong to one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, Multi-language types, or Byte as specified in Section 5.9.2. The origin and result buffers (origin_addr and result_addr) must be disjoint.

### 11.3.5 Request-based RMA Communication Operations

Request-based RMA communication operations allow the user to associate a request handle with the RMA operations and test or wait for the completion of these requests using the functions described in Section 3.7.3. Request-based RMA operations are only valid within a passive target epoch (see Section 11.5).

Upon returning from a completion call in which an RMA operation completes, the MPI_ERROR field in the associated status object is set appropriately (see Section 3.2.5). All other fields of status and the results of status query functions (e.g., MPI_GET_COUNT) are undefined. It is valid to mix different request types (e.g., any combination of RMA requests, collective requests, I/O requests, generalized requests, or point-to-point requests) in functions that enable multiple completions (e.g., MPI_WAITALL). It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for a request associated with an RMA operation. RMA requests are not persistent.

The end of the epoch, or explicit bulk synchronization using MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, or MPI_WIN_FLUSH_LOCAL_ALL, also indicates completion of the RMA operations. However, users must still wait or test on the request handle to allow the MPI implementation to clean up any resources associated with these requests; in such cases the wait operation will complete locally.

MPI_RPUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

| | | |
|---|---|---|
| IN | origin_addr | initial address of origin buffer (choice) |
| IN | origin_count | number of entries in origin buffer (non-negative integer) |
| IN | origin_datatype | datatype of each entry in origin buffer (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | win | window object used for communication (handle) |
| OUT | request | RMA request (handle) |

**C binding**

```
int MPI_Rput(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
              TARGET_DATATYPE, WIN, REQUEST, IERROR
```

MPI_RPUT is similar to MPI_PUT (Section 11.3.1), except that it allocates a communication request object and associates it with the request handle (the argument request).

The completion of an MPI_RPUT operation (i.e., after the corresponding test or wait) indicates that the sender is now free to update the locations in the origin buffer. It does not indicate that the data is available at the target window. If remote completion is required, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL can be used.

MPI_RGET(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win, request)

| | | |
|---|---|---|
| OUT | origin_addr | initial address of origin buffer (choice) |
| IN | origin_count | number of entries in origin buffer (non-negative integer) |
| IN | origin_datatype | datatype of each entry in origin buffer (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from window start to the beginning of the target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | win | window object used for communication (handle) |
| OUT | request | RMA request (handle) |

**C binding**

```
int MPI_Rget(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_Request *request)
```

**F08 binding**

```
MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, request,
             ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*)
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
              TARGET_DATATYPE, WIN, REQUEST, IERROR
```

MPI_RGET is similar to MPI_GET (Section 11.3.2), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RGET operation indicates that the data is available in the origin buffer. If origin_addr points to memory attached to a window, then the data becomes available in the private copy of this window.

MPI_RACCUMULATE(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, op, win, request)

| | | |
|---|---|---|
| IN | origin_addr | initial address of buffer (choice) |
| IN | origin_count | number of entries in buffer (non-negative integer) |
| IN | origin_datatype | datatype of each entry in origin buffer (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to beginning of target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | op | reduce operation (handle) |
| IN | win | window object (handle) |
| OUT | request | RMA request (handle) |

### C binding

```
int MPI_Raccumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
             MPI_Request *request)
```

### F08 binding

```
MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
              TARGET_DATATYPE, OP, WIN, REQUEST, IERROR
```

MPI_RACCUMULATE is similar to MPI_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RACCUMULATE operation indicates that the origin buffer is free to be updated. It does not indicate that the operation has completed at the target window.

MPI_RGET_ACCUMULATE(origin_addr, origin_count, origin_datatype, result_addr, result_count, result_datatype, target_rank, target_disp, target_count, target_datatype, op, win, request)

| | | |
|---|---|---|
| IN | origin_addr | initial address of buffer (choice) |
| IN | origin_count | number of entries in origin buffer (non-negative integer) |
| IN | origin_datatype | datatype of each entry in origin buffer (handle) |
| OUT | result_addr | initial address of result buffer (choice) |
| IN | result_count | number of entries in result buffer (non-negative integer) |
| IN | result_datatype | datatype of each entry in result buffer (handle) |
| IN | target_rank | rank of target (non-negative integer) |
| IN | target_disp | displacement from start of window to beginning of target buffer (non-negative integer) |
| IN | target_count | number of entries in target buffer (non-negative integer) |
| IN | target_datatype | datatype of each entry in target buffer (handle) |
| IN | op | reduce operation (handle) |
| IN | win | window object (handle) |
| OUT | request | RMA request (handle) |

**C binding**

```
int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, void *result_addr,
             int result_count, MPI_Datatype result_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
             MPI_Request *request)
```

**F08 binding**

```
MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
             result_addr, result_count, result_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    INTEGER, INTENT(IN) ::  origin_count, result_count, target_rank,
              target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype,
              result_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
             RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
    IERROR
```

MPI_RGET_ACCUMULATE is similar to MPI_GET_ACCUMULATE (Section 11.3.4), except that it allocates a communication request object and associates it with the request handle (the argument request) that can be used to wait or test for completion. The completion of an MPI_RGET_ACCUMULATE operation indicates that the data is available in the result buffer and the origin buffer is free to be updated. It does not indicate that the operation has been completed at the target window.

## 11.4 Memory Model

The memory semantics of RMA are best understood by using the concept of *public* and *private* window copies. We assume that systems have a public memory region that is addressable by all processes (e.g., the shared memory in shared memory machines or the exposed main memory in distributed memory machines). In addition, most machines have fast private buffers (e.g., transparent caches or explicit communication buffers) local to each process where copies of data elements from the main memory can be stored for faster access. Such buffers are either coherent, i.e., all updates to main memory are reflected in all private copies consistently, or non-coherent, i.e., conflicting accesses to main memory need to be synchronized and updated in all private copies explicitly. Coherent systems allow direct updates to remote memory without any participation of the remote side. Non-coherent systems, however, need to call RMA functions in order to reflect updates to the

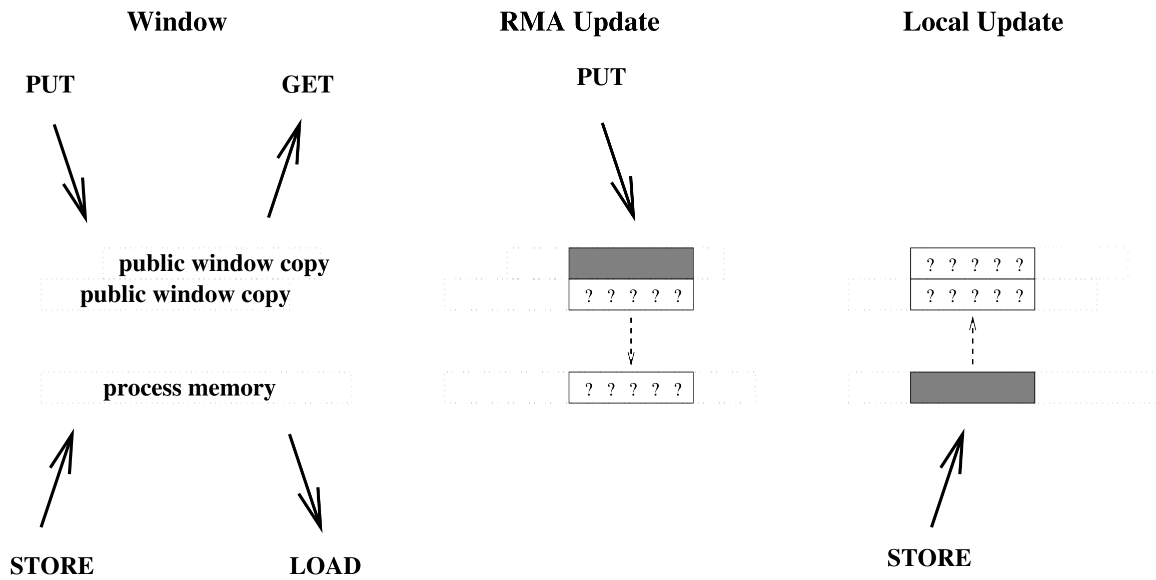


Figure 11.1: Schematic description of the public/private window operations in the MPI_WIN_SEPARATE memory model for two overlapping windows.

public window in their private memory. Thus, in coherent memory, the public and the private window are identical while they remain logically separate in the non-coherent case. MPI thus differentiates between two **memory models** called **RMA unified**, if public and private window are logically identical, and **RMA separate**, otherwise.

In the RMA separate model, 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 local store accesses and updates the instance in process memory (this includes MPI receives), but the update may affect other public copies of the same locations. A get on a window accesses the public copy of that window. A put or accumulate on a window accesses and updates the public copy of that window, but the update may affect the private copy of the same locations in process memory, and public copies of other overlapping windows. This is illustrated in Figure 11.1.

In the RMA unified model, public and private copies are identical and updates via put or accumulate calls are eventually observed by load operations without additional RMA calls. A store access to a window is eventually visible to remote get or accumulate calls without additional RMA calls. These stronger semantics of the RMA unified model allow the user to omit some synchronization calls and potentially improve performance.

> *Advice to users.* If accesses in the RMA unified model are not synchronized (with locks or flushes, see Section 11.5.3), load and store operations might observe changes to the memory while they are in progress. The order in which data is written is not specified unless further synchronization is used. This might lead to inconsistent views on memory and programs that assume that a transfer is complete by only checking parts of the message are erroneous. (*End of advice to users.*)

The memory model for a particular RMA window can be determined by accessing the attribute MPI_WIN_MODEL. If the memory model is the unified model, the value of this attribute is MPI_WIN_UNIFIED; otherwise, the value is MPI_WIN_SEPARATE.

## 11.5 Synchronization Calls

RMA communications fall in two categories:

- **active target communication**, where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the communication. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization.
- **passive target communication**, where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location.

RMA communication calls with argument win must occur at a process only within an **access epoch** for win. Such an epoch starts with an RMA synchronization call on win; it proceeds with zero or more RMA communication calls (e.g., MPI_PUT, MPI_GET or MPI_ACCUMULATE) on win; it completes with another synchronization call on win. This allows users to amortize one synchronization with multiple data transfers and provide implementors more flexibility in the implementation of RMA operations.

Distinct access epochs for win at the same process must be disjoint. On the other hand, epochs pertaining to different win arguments may overlap. Local operations or other MPI calls may also occur during an epoch.

In active target communication, a target window can be accessed by RMA operations only within an **exposure epoch**. Such an epoch is started and completed by RMA synchronization calls executed by the target process. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with exposure epochs on other windows or with access epochs for the same or other win arguments. There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

In passive target communication the target process does not execute RMA synchronization calls, and there is no concept of an exposure epoch.

MPI provides three synchronization mechanisms:

1. The MPI_WIN_FENCE collective synchronization call supports a simple synchronization pattern that is often used in parallel computations: namely a loosely-synchronous model, where global computation phases alternate with global communication phases. This mechanism is most useful for loosely synchronous algorithms where the graph of communicating processes changes very frequently, or where each process communicates with many others.

   This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_WIN_FENCE. A process can access windows at all processes in the group of win during such an access epoch, and the local window can be accessed by all processes in the group of win during such an exposure epoch.

2. The four functions MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST, and MPI_WIN_WAIT can be used to restrict synchronization to the minimum: only pairs of communicating processes synchronize, and they do so only when a synchronization is needed to order correctly 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 MPI_WIN_START and is terminated by a call to MPI_WIN_COMPLETE. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to MPI_WIN_POST and is completed by a call to MPI_WIN_WAIT. The post call has a group argument that specifies the set of origin processes for that epoch.

3. Finally, shared lock access is provided by the functions MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, MPI_WIN_UNLOCK, and MPI_WIN_UNLOCK_ALL. MPI_WIN_LOCK and MPI_WIN_UNLOCK also provide exclusive lock capability. 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 four calls provide passive target communication. An access epoch is started by a call to MPI_WIN_LOCK or MPI_WIN_LOCK_ALL and terminated by a call to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL, respectively.

Figure 11.2 illustrates the general synchronization pattern for active target communication. The synchronization between `post` and `start` ensures that the put call of the origin process does not start until the target process exposes the window (with the `post` call); the target process will expose the window only after preceding local accesses to the window have completed. The synchronization between `complete` and `wait` ensures that the put call of the origin process completes before the window is unexposed (with the `wait` call). The target process will execute following local accesses to the target window only after the `wait` returned.

Figure 11.2 shows operations occurring in the natural temporal order implied by the synchronizations: the `post` occurs before the matching `start`, and `complete` occurs before the matching `wait`. However, such **strong synchronization** is more than needed for correct ordering of window accesses. The semantics of MPI calls allow **weak synchronization**, as illustrated in Figure 11.3. The access to the target window is delayed until the window is exposed, after the `post`. However the `start` may complete earlier; the `put` and `complete` may also terminate earlier, if put data is buffered by the implementation. The synchronization calls order correctly window accesses, but do not necessarily synchronize other operations. This weaker synchronization semantic allows for more efficient implementations.

Figure 11.4 illustrates the general synchronization pattern for passive target communication. The first origin process communicates data to the second origin process, through the memory of the target process; the target process is not explicitly involved in the communication. The `lock` and `unlock` calls ensure that the two RMA accesses do not occur

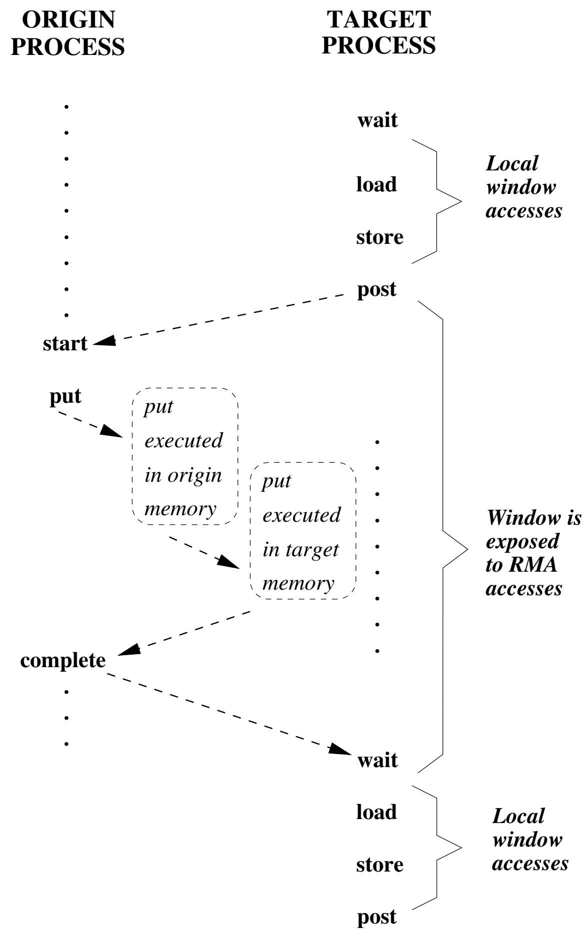


Figure 11.2: Active target communication. Dashed arrows represent synchronizations (ordering of events).

concurrently. However, they do *not* ensure that the `put` by origin 1 will precede the `get` by origin 2.

> *Rationale.* RMA does not define fine-grained mutexes in memory (only logical coarse-grained process locks). MPI provides the primitives (compare and swap, accumulate, send/receive, etc.) needed to implement high-level synchronization operations. (*End of rationale.*)

### 11.5.1 Fence

MPI_WIN_FENCE(assert, win)

| | | |
|---|---|---|
| IN | assert | program assertion (integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_fence(int assert, MPI_Win win)
```

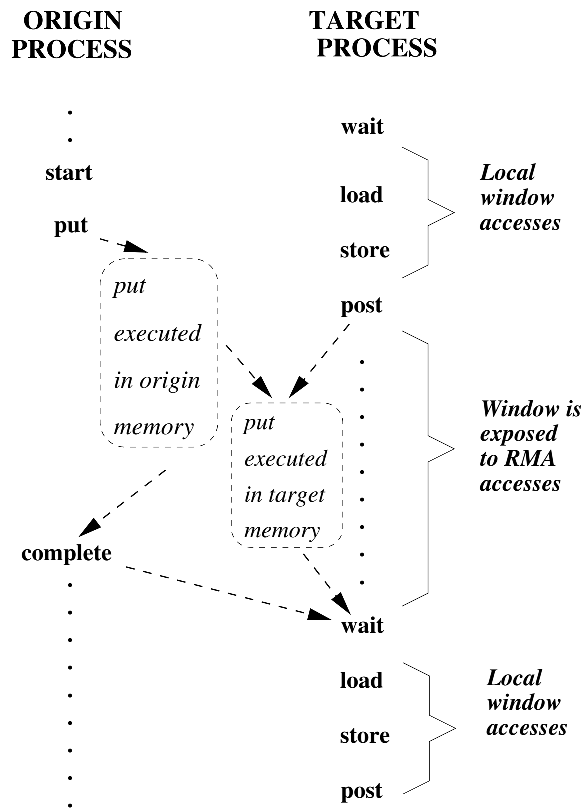


Figure 11.3: Active target communication, with weak synchronization. Dashed arrows represent synchronizations (ordering of events)

**F08 binding**

```
MPI_Win_fence(assert, win, ierror)
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

The MPI call MPI_WIN_FENCE(assert, win) synchronizes RMA calls on win. The call is collective on the group of win. All RMA operations on win originating at a given process and started before the fence call will complete at that process before the fence call returns. They will be completed at their target before the fence call returns at the target. RMA operations on win started by a process after the fence call returns will access their target window only after MPI_WIN_FENCE has been called by the target process.

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

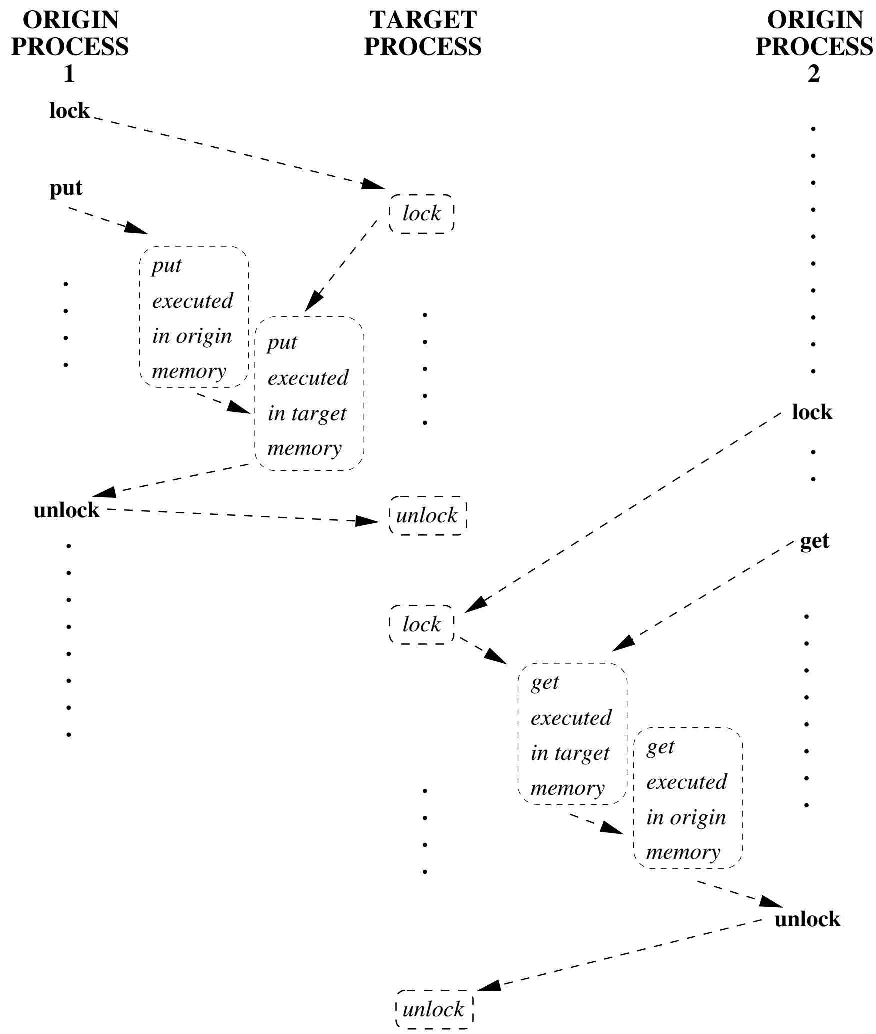


Figure 11.4: Passive target communication. Dashed arrows represent synchronizations (ordering of events).

fence call and the local window is the target of RMA accesses between these two fence calls. Thus, the fence call is equivalent to calls to a subset of `post, start, complete, wait`.

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

The assert argument is used to provide assertions on the context of the call that may be used for various optimizations. This is described in Section 11.5.5. A value of assert = 0 is always valid.

*Advice to users.* Calls to MPI_WIN_FENCE should both precede and follow calls to RMA communication functions that are synchronized with fence calls. (*End of advice to users.*)

### 11.5.2 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 binding**

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

**F08 binding**

```
MPI_Win_start(group, assert, win, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

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 11.5.5. A value of assert = 0 is always valid.

MPI_WIN_COMPLETE(win)

| | | |
|---|---|---|
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_complete(MPI_Win win)
```

**F08 binding**

```
MPI_Win_complete(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_COMPLETE(WIN, IERROR)
    INTEGER WIN, IERROR
```

Completes an RMA access epoch on win started by a call to MPI_WIN_START. All RMA communication calls issued on win during this epoch will have completed at the origin when the call returns.

MPI_WIN_COMPLETE enforces completion of preceding RMA calls at the origin, but not at the target. A put or accumulate call may not have completed at the target when it has completed at the origin.

Consider the sequence of calls in the example below.

**Example 11.4**

```
MPI_Win_start(group, flag, win);
MPI_Put(..., win);
MPI_Win_complete(win);
```

The call to MPI_WIN_COMPLETE does not return until the put call has completed at the origin; and the target window will be accessed by the put operation only after the call to MPI_WIN_START has matched a call to MPI_WIN_POST by the target process. This still leaves much choice to implementors. The call to MPI_WIN_START can block until the matching call to MPI_WIN_POST occurs at all target processes. One can also have implementations where the call to MPI_WIN_START is nonblocking, but the call to MPI_PUT blocks until the matching call to MPI_WIN_POST occurs; or implementations where the first two calls are nonblocking, but the call to MPI_WIN_COMPLETE blocks until the call to MPI_WIN_POST occurred; or even implementations where all three calls can complete before any target process has called MPI_WIN_POST — the data put must be buffered, in this last case, so as to allow the put to complete at the origin ahead of its completion at the target. However, once the call to MPI_WIN_POST is issued, the sequence above must complete, without further dependencies.

MPI_WIN_POST(group, assert, win)

| | | |
|---|---|---|
| IN | group | group of origin processes (handle) |
| IN | assert | program assertion (integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

**F08 binding**

```
MPI_Win_post(group, assert, win, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

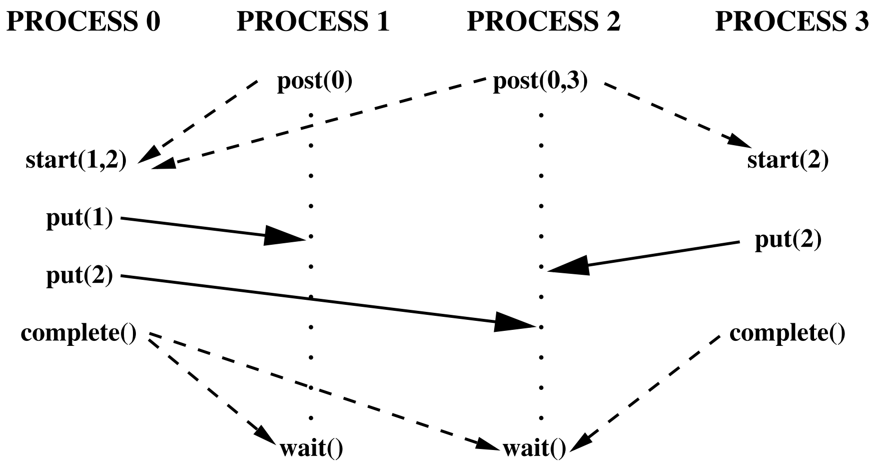


Figure 11.5: Active target communication. Dashed arrows represent synchronizations and solid arrows represent data transfer.

Starts an RMA exposure epoch for the local window associated with win. Only processes in group should access the window with RMA calls on win during this epoch. Each process in group must issue a matching call to MPI_WIN_START. MPI_WIN_POST does not block.

MPI_WIN_WAIT(win)

| | | |
|---|---|---|
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_wait(MPI_Win win)
```

**F08 binding**

```
MPI_Win_wait(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

Completes an RMA exposure epoch started by a call to MPI_WIN_POST on win. This call matches calls to MPI_WIN_COMPLETE(win) issued by each of the origin processes that were granted access to the window during this epoch. The call to MPI_WIN_WAIT will block until all matching calls to MPI_WIN_COMPLETE have occurred. This guarantees that all these origin processes have completed their RMA accesses to the local window. When the call returns, all these RMA accesses will have completed at the target window.

Figure 11.5 illustrates the use of these four functions. Process 0 puts data in the windows of processes 1 and 2 and process 3 puts data in the window of process 2. Each start call lists the ranks of the processes whose windows will be accessed; each post call lists the ranks of the processes that access the local window. The figure illustrates a possible timing for the events, assuming strong synchronization; in a weak synchronization, the start, 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) |

**C binding**

```
int MPI_Win_test(MPI_Win win, int *flag)
```

**F08 binding**

```
MPI_Win_test(win, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_TEST(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
```

This is the nonblocking version of MPI_WIN_WAIT. It returns flag = true if all accesses to the local window by the group to which it was exposed by the corresponding MPI_WIN_POST call have been completed as signalled by matching MPI_WIN_COMPLETE calls, and flag = false otherwise. In the former case MPI_WIN_WAIT would have returned immediately. The effect of return of MPI_WIN_TEST with flag = true is the same as the effect of a return of MPI_WIN_WAIT. If flag = false is returned, then the call has no visible effect.

MPI_WIN_TEST should be invoked only where MPI_WIN_WAIT can be invoked. Once the call has returned flag = true, it must not be invoked anew, until the window is posted anew.

Assume that window win is associated with a "hidden" communicator wincomm, used for communication by the processes of win. The rules for matching of post and start calls and for matching complete and wait calls can be derived from the rules for matching sends and receives, by considering the following (partial) model implementation.

**MPI_WIN_POST(group,0,win)** initiates a nonblocking send with tag tag0 to each process in group, using wincomm. There is no need to wait for the completion of these sends.

**MPI_WIN_START(group,0,win)** initiates a nonblocking receive with tag tag0 from each process in group, using wincomm. An RMA access to a window in target process i is delayed until the receive from i is completed.

**MPI_WIN_COMPLETE(win)** initiates a nonblocking send with tag tag1 to each process in the group of the preceding start call. No need to wait for the completion of these sends.

**MPI_WIN_WAIT(win)** initiates a nonblocking receive with tag tag1 from each process in the group of the preceding post call. Wait for the completion of all receives.

No races can occur in a correct program: each of the sends matches a unique receive, and vice versa.

*Rationale.* The design for general active target synchronization requires the user to provide complete information on the communication pattern, at each end of a communication link: each origin specifies a list of targets, and each target specifies a list of origins. This provides maximum flexibility (hence, efficiency) for the implementor: each synchronization can be initiated by either side, since each "knows" the identity of the other. This also provides maximum protection from possible races. On the other hand, the design requires more information than RMA needs: in general, it is sufficient for the origin to know the rank of the target, but not vice versa. Users that want more "anonymous" communication will be required to use the fence or lock mechanisms. (*End of rationale.*)

*Advice to users.* Assume a communication pattern that is represented by a directed graph $G = \langle V, E \rangle$, where $V = \{0, \ldots, n-1\}$ 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($ingroup_i$, ...), followed by a call to MPI_WIN_START($outgroup_i$,...), where $outgroup_i = \{j \ : \ ij \in E\}$ and $ingroup_i = \{j \ : \ ji \in E\}$. A call is a noop, and can be skipped, if the group argument is empty. After the communications calls, each process that issued a start will issue a complete. Finally, each process that issued a post will issue a wait.

Note that each process may call with a group argument that has different members. (*End of advice to users.*)

### 11.5.3 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 (non-negative integer) |
| IN | assert | program assertion (integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
```

**F08 binding**

```
MPI_Win_lock(lock_type, rank, assert, win, ierror)
    INTEGER, INTENT(IN) ::  lock_type, rank, assert
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR
```

Starts an RMA access epoch. The window at the process with rank rank can be accessed by RMA operations on win during that epoch. Multiple RMA access epochs (with calls to MPI_WIN_LOCK) can occur simultaneously; however, each access epoch must target a different process.

MPI_WIN_LOCK_ALL(assert, win)

| | | |
|---|---|---|
| IN | assert | program assertion (integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_lock_all(int assert, MPI_Win win)
```

**F08 binding**

```
MPI_Win_lock_all(assert, win, ierror)
    INTEGER, INTENT(IN) :: assert
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR
```

Starts an RMA access epoch to all processes in win, with a lock type of MPI_LOCK_SHARED. During the epoch, the calling process can access the window memory on all processes in win by using RMA operations. A window locked with MPI_WIN_LOCK_ALL must be unlocked with MPI_WIN_UNLOCK_ALL. This routine is not collective — the ALL refers to a lock on all members of the group of the window.

*Advice to users.* There may be additional overheads associated with using MPI_WIN_LOCK and MPI_WIN_LOCK_ALL concurrently on the same window. These overheads could be avoided by specifying the assertion MPI_MODE_NOCHECK when possible (see Section 11.5.5). (*End of advice to users.*)

MPI_WIN_UNLOCK(rank, win)

| | | |
|---|---|---|
| IN | rank | rank of window (non-negative integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_unlock(int rank, MPI_Win win)
```

**F08 binding**

```
MPI_Win_unlock(rank, win, ierror)
    INTEGER, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_UNLOCK(RANK, WIN, IERROR)
```

```
    INTEGER RANK, WIN, IERROR
```

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

MPI_WIN_UNLOCK_ALL(win)

| | | |
|---|---|---|
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_unlock_all(MPI_Win win)
```

**F08 binding**

```
MPI_Win_unlock_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_WIN_UNLOCK_ALL(WIN, IERROR)
    INTEGER WIN, IERROR
```

Completes a shared RMA access epoch started by a call to MPI_WIN_LOCK_ALL on window win. RMA operations issued during this epoch 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 calls, and to protect load/store accesses to a locked local or shared memory window executed between the lock and unlock calls. 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. For example, a process may not call MPI_WIN_LOCK to lock a target window if the target process has called MPI_WIN_POST and has not yet called MPI_WIN_WAIT; it is erroneous to call MPI_WIN_POST while the local window is locked.

> *Rationale.* An alternative is to require MPI to enforce mutual exclusion between exposure epochs and locking periods. But this would entail additional overheads when locks or active target synchronization do not interact in support of those rare interactions between the two mechanisms. The programming style that we encourage here is that a set of windows is used with only one synchronization mechanism at a time, with shifts from one mechanism to another being rare and involving global synchronization. (*End of rationale.*)

> *Advice to users.* Users need to use explicit synchronization code in order to enforce mutual exclusion between locking periods and exposure epochs on a window. (*End of advice to users.*)

Implementors may restrict the use of RMA communication that is synchronized by lock calls to windows in memory allocated by MPI_ALLOC_MEM (Section 8.2), MPI_WIN_ALLOCATE (Section 11.2.2), MPI_WIN_ALLOCATE_SHARED (Section 11.2.3), or attached with MPI_WIN_ATTACH (Section 11.2.4). Locks can be used portably only in such memory.

*Rationale.* The implementation of passive target communication when memory is not shared may require an asynchronous software agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for third party communication in shared memory machines.

(*End of rationale.*)

Consider the sequence of calls in the example below.

**Example 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 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.

### 11.5.4 Flush and Sync

All flush and sync functions can be called only within passive target epochs.

MPI_WIN_FLUSH(rank, win)

| | | |
|---|---|---|
| IN | rank | rank of target window (non-negative integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_flush(int rank, MPI_Win win)
```

**F08 binding**

```
MPI_Win_flush(rank, win, ierror)
    INTEGER, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_FLUSH(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
```

MPI_WIN_FLUSH completes all outstanding RMA operations initiated by the calling process to the target rank on the specified window. The operations are completed both at the origin and at the target.

MPI_WIN_FLUSH_ALL(win)

| | | |
|---|---|---|
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_flush_all(MPI_Win win)
```

**F08 binding**

```
MPI_Win_flush_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_FLUSH_ALL(WIN, IERROR)
    INTEGER WIN, IERROR
```

All RMA operations issued by the calling process to any target on the specified window prior to this call and in the specified window will have completed both at the origin and at the target when this call returns.

MPI_WIN_FLUSH_LOCAL(rank, win)

| | | |
|---|---|---|
| IN | rank | rank of target window (non-negative integer) |
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_flush_local(int rank, MPI_Win win)
```

**F08 binding**

```
MPI_Win_flush_local(rank, win, ierror)
    INTEGER, INTENT(IN) :: rank
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR
```

Locally completes at the origin all outstanding RMA operations initiated by the calling process to the target process specified by rank on the specified window. For example, after this routine completes, the user may reuse any buffers provided to put, get, or accumulate operations.

MPI_WIN_FLUSH_LOCAL_ALL(win)

| | | |
|---|---|---|
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_flush_local_all(MPI_Win win)
```

**F08 binding**

```
MPI_Win_flush_local_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
    INTEGER WIN, IERROR
```

All RMA operations issued to any target prior to this call in this window will have completed at the origin when MPI_WIN_FLUSH_LOCAL_ALL returns.

MPI_WIN_SYNC(win)

| | | |
|---|---|---|
| IN | win | window object (handle) |

**C binding**

```
int MPI_Win_sync(MPI_Win win)
```

**F08 binding**

```
MPI_Win_sync(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_WIN_SYNC(WIN, IERROR)
    INTEGER WIN, IERROR
```

The call MPI_WIN_SYNC synchronizes the private and public window copies of win. For the purposes of synchronizing the private and public window, MPI_WIN_SYNC has the effect of ending and reopening an access and exposure epoch on the window (note that it does not actually end an epoch or complete any pending MPI RMA operations).

### 11.5.5 Assertions

The assert argument in the calls MPI_WIN_POST, MPI_WIN_START, MPI_WIN_FENCE, MPI_WIN_LOCK, and MPI_WIN_LOCK_ALL is used to provide assertions on the context of the call that may be used to optimize performance. The assert argument does not change program semantics if it provides correct information on the program — it is erroneous to provide incorrect information. Users may always provide assert = 0 to indicate a general case where no guarantees are made.

*Advice to users.* Many implementations may not take advantage of the information in assert; some of the information is relevant only for noncoherent shared memory machines. Users should consult their implementation's manual to find which information is useful on each system. On the other hand, applications that provide correct assertions whenever applicable are portable and will take advantage of assertion specific optimizations whenever available. (*End of advice to users.*)

*Advice to implementors.* Implementations can always ignore the assert argument. Implementors should document which assert values are significant on their implementation. (*End of advice to implementors.*)

assert is the bit-vector OR of zero or more of the following integer constants: 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. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition!). (*End of advice to users.*)

**MPI_WIN_START:**

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_POST have already completed on all target processes when the call to MPI_WIN_START is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)

**MPI_WIN_POST:**

MPI_MODE_NOCHECK — the matching calls to MPI_WIN_START have not yet occurred on any origin processes when the call to MPI_WIN_POST is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.

MPI_MODE_NOSTORE — the local window was not updated by stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.

MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.

**MPI_WIN_FENCE:**

MPI_MODE_NOSTORE — the local window was not updated by stores (or local get or receive calls) since last synchronization.

MPI_MODE_NOPUT — the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.

MPI_MODE_NOPRECEDE — the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.

MPI_MODE_NOSUCCEED — the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.

**MPI_WIN_LOCK, MPI_WIN_LOCK_ALL:**

MPI_MODE_NOCHECK — no other process holds, or will attempt to acquire, a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

*Advice to users.* Note that the nostore and noprecede flags provide information on what happened *before* the call; the noput and nosucceed flags provide information on what will happen *after* the call. (*End of advice to users.*)

### 11.5.6 Miscellaneous Clarifications

Once an RMA routine completes, it is safe to free any opaque objects passed as arguments to that routine. For example, the datatype argument of a MPI_PUT call can be freed as soon as the call returns, even though the communication may not be complete.

As in message-passing, datatypes must be committed before they can be used in RMA communication.

## 11.6 Error Handling

### 11.6.1 Error Handlers

Errors occurring during calls to routines that create MPI windows (e.g., MPI_WIN_CREATE (...,comm,...)) cause the error handler currently associated with comm to be invoked. All other RMA calls have an input win argument. When an error occurs during such a call, the error handler currently associated with win is invoked.

The 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).

### 11.6.2 Error Classes

The error classes for one-sided communication are defined in Table 11.2. RMA routines may (and almost certainly will) use other MPI error classes, such as MPI_ERR_OP or MPI_ERR_RANK.

## 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

| | |
|---|---|
| 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 | invalid synchronization of RMA calls |
| MPI_ERR_RMA_RANGE | target memory is not part of the window (in the case of a window created with MPI_WIN_CREATE_DYNAMIC, target memory is not attached) |
| MPI_ERR_RMA_ATTACH | memory cannot be attached (e.g., because of resource exhaustion) |
| MPI_ERR_RMA_SHARED | memory cannot be shared (e.g., some process in the group of the specified communicator cannot expose shared memory) |
| MPI_ERR_RMA_FLAVOR | passed window has the wrong flavor for the called function |

Table 11.2: Error classes in one-sided communication routines

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, MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL, MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL, MPI_WIN_UNLOCK, or MPI_WIN_UNLOCK_ALL that synchronizes this access at the origin.

2. If an RMA operation is completed at the origin by a call to MPI_WIN_FENCE then the operation is completed at the target by the matching call to MPI_WIN_FENCE by the target process.

3. If an RMA operation is completed at the origin by a call to MPI_WIN_COMPLETE then the operation is completed at the target by the matching call to MPI_WIN_WAIT by the target process.

4. If an RMA operation is completed at the origin by a call to MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, MPI_WIN_FLUSH(rank=target), or MPI_WIN_FLUSH_ALL, then the operation is completed at the target by that same call.

5. An update of a location in a private window copy in process memory becomes visible in the public window copy at latest when an ensuing call to MPI_WIN_POST, MPI_WIN_FENCE, MPI_WIN_UNLOCK, MPI_WIN_UNLOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA

unified memory model, an update of a location in a private window in process memory becomes visible without additional RMA calls.

6. An update by a put or accumulate call to a public window copy becomes visible in the private copy in process memory at latest when an ensuing call to MPI_WIN_WAIT, MPI_WIN_FENCE, MPI_WIN_LOCK, MPI_WIN_LOCK_ALL, or MPI_WIN_SYNC is executed on that window by the window owner. In the RMA unified memory model, an update by a put or accumulate call to a public window copy eventually becomes visible in the private copy in process memory without additional RMA calls.

The MPI_WIN_FENCE or MPI_WIN_WAIT call that completes the transfer from public copy to private copy (6) is the same call that completes the put or accumulate operation in the window copy (2, 3). If a put or accumulate access was synchronized with a lock, then the update of the public window copy is complete as soon as the updating process executed MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL. In the RMA separate memory model, the update of a private copy in the process memory may be delayed until the target process executes a synchronization call on that window (6). Thus, updates to process memory can always be delayed in the RMA separate memory model until the process executes a suitable synchronization call, while they must complete in the RMA unified model without additional synchronization calls. If fence or post-start-complete-wait synchronization is used, updates to a public window copy can be delayed in both memory models until the window owner executes a synchronization call. When passive target synchronization is used, it is necessary to update the public window copy even if the window owner does not execute any related synchronization call.

The rules above also define, by implication, when an update to a public window copy becomes visible in another overlapping public window copy. Consider, for example, two overlapping windows, win1 and win2. A call to MPI_WIN_FENCE(0, win1) by the window owner makes visible in the process memory previous updates to window win1 by remote processes. A subsequent call to MPI_WIN_FENCE(0, win2) makes these updates visible in the public copy of win2.

The behavior of some MPI RMA operations may be *undefined* in certain situations. For example, the result of several origin processes performing concurrent MPI_PUT operations to the same target location is undefined. In addition, the result of a single origin process performing multiple MPI_PUT operations to the same target location within the same access epoch is also undefined. The result at the target may have all of the data from one of the MPI_PUT operations (the "last" one, in some sense), bytes from some of each of the operations, or something else. In MPI-2, such operations were *erroneous*. That meant that an MPI implementation was permitted to signal an MPI exception. Thus, user programs or tools that used MPI RMA could not portably permit such operations, even if the application code could function correctly with such an undefined result. In MPI-3, these operations are not erroneous, but do not have a defined behavior.

*Rationale.* As discussed in [6], requiring operations such as overlapping puts to be erroneous makes it difficult to use MPI RMA to implement programming models—such as Unified Parallel C (UPC) or SHMEM—that permit these operations. Further, while MPI-2 defined these operations as erroneous, the MPI Forum is unaware of any implementation that enforces this rule, as it would require significant overhead. Thus, relaxing this condition does not impact existing implementations or applications. (*End of rationale.*)

*Advice to implementors.* Overlapping accesses are undefined. However, to assist users in debugging code, implementations may wish to provide a mode in which such operations are detected and reported to the user. Note, however, that in MPI-3, such operations must not generate an MPI exception. (*End of advice to implementors.*)

A program with a well-defined outcome in the MPI_WIN_SEPARATE memory model must obey the following rules.

S1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update becomes visible in the private window copy in process memory.

S2. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy. There is one exception to this rule, in the case where the same variable is updated by two concurrent accumulates with the same predefined datatype, on the same window. Additional restrictions on the operation apply, see the info key accumulate_ops in Section 11.2.1.

S3. A put or accumulate must not access a target window once a store or a put or accumulate update to another (overlapping) target window has started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a store to 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.

*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 updated by stores and for locations that were remotely updated by put or accumulate operations. Without this constraint, the MPI library would 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.*)

Note that MPI_WIN_SYNC may be used within a passive target epoch to synchronize the private and public window copies (that is, updates to one are made visible to the other).

In the MPI_WIN_UNIFIED memory model, the rules are simpler because the public and private windows are the same. However, there are restrictions to avoid concurrent access to the same memory locations by different processes. The rules that a program with a well-defined outcome must obey in this case are:

U1. A location in a window must not be accessed with load/store operations once an update to that location has started, until the update is complete, subject to the following special case.

U2. Accessing a location in the window that is also the target of a remote update is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Updates from a remote process will appear in the memory of the target, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory of the target, the data remains until replaced by another update. This permits polling on a location for a change from zero to non-zero or for a particular value, but not polling and comparing the relative magnitude of values. Users are cautioned that polling on one memory location and then accessing a different memory location has defined behavior only if the other rules given here and in this chapter are followed.

*Advice to users.* Some compiler optimizations can result in code that maintains the sequential semantics of the program, but violates this rule by introducing temporary values into locations in memory. Most compilers only apply such transformations under very high levels of optimization and users should be aware that such aggressive optimization may produce unexpected results. (*End of advice to users.*)

U3. Updating a location in the window with a store operation that is also the target of a remote read (but not update) is valid (not erroneous) but the precise result will depend on the behavior of the implementation. Store updates will appear in memory, but there are no atomicity or ordering guarantees if more than one byte is updated. Updates are stable in the sense that once data appears in memory, the data remains until replaced by another update. This permits updates to memory with store operations without requiring an RMA epoch. Users are cautioned that remote accesses to a window that is updated by the local process has defined behavior only if the other rules given here and elsewhere in this chapter are followed.

U4. A location in a window must not be accessed as a target of an RMA operation once an update to that location has started and until the update completes at the target. There is one exception to this rule: in the case where the same location is updated by two concurrent accumulates with the same predefined datatype on the same window. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 11.2.1.

U5. A put or accumulate must not access a target window once a store, put, or accumulate update to another (overlapping) target window has started on the same location in the target window and until the update completes at the target window. Conversely, a store operation to a location in a window must not start once a put or accumulate update to the same location in that target window has started and until the put or accumulate update completes at the target.

*Advice to users.* In the unified memory model, in the case where the window is in shared memory, MPI_WIN_SYNC can be used to order store operations and make store updates to the window visible to other processes and threads. Use of this routine is necessary to ensure portable behavior when point-to-point, collective, or shared memory synchronization is used in place of an RMA synchronization routine. MPI_WIN_SYNC should be called by the writer before the non-RMA synchronization operation and by the reader after the non-RMA synchronization, as shown in Example 11.21. (*End of advice to users.*)

A program that violates these rules has undefined behavior.

*Advice to users.* A user can write correct programs by following the following rules:

**fence:** During each period between fence calls, each window is either updated by put or accumulate calls, or updated by stores, but not both. Locations updated by put or accumulate calls should not be accessed during the same period (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated during the same period.

**post-start-complete-wait:** A window should not be updated with store operations while posted if it is being updated by put or accumulate calls. Locations updated by put or accumulate calls should not be accessed while the window is posted (with the exception of concurrent updates to the same location by accumulate calls). Locations accessed by get calls should not be updated while the window is posted.

With the post-start synchronization, the target process can tell the origin process that its window is now ready for RMA access; with the complete-wait synchronization, the origin process can tell the target process that it has finished its RMA accesses to the window.

**lock:** Updates to the window are protected by exclusive locks if they may conflict. Nonconflicting accesses (such as read-only accesses or accumulate accesses) are protected by shared locks, both for load/store accesses and for RMA accesses.

**changing window or synchronization mode:** One can change synchronization mode, or change the window used to access a location that belongs to two overlapping windows, when the process memory and the window copy are guaranteed to have the same values. This is true after a local call to MPI_WIN_FENCE, if RMA accesses to the window are synchronized with fences; after a local call to MPI_WIN_WAIT, if the accesses are synchronized with post-start-complete-wait; after the call at the origin (local or remote) to MPI_WIN_UNLOCK or MPI_WIN_UNLOCK_ALL if the accesses are synchronized with locks.

In addition, a process should not access the local buffer of a get operation until the operation is complete, and should not update the local buffer of a put or accumulate operation until that operation is complete.

The RMA synchronization operations define when updates are guaranteed to become visible in public and private windows. Updates may become visible earlier, but such behavior is implementation dependent. (*End of advice to users.*)

The semantics are illustrated by the following examples:

**Example 11.6** The following example demonstrates updating a memory location inside a window for the separate memory model, according to Rule 5. The MPI_WIN_LOCK and MPI_WIN_UNLOCK calls around the store to `X` in process B are necessary to ensure consistency between the public and private copies of the window.

```
Process A:                    Process B:
                              window location X

                              MPI_Win_lock(EXCLUSIVE, B)
                              store X /* local update to private copy of B */
                              MPI_Win_unlock(B)
                              /* now visible in public window copy */

MPI_Barrier                   MPI_Barrier

MPI_Win_lock(EXCLUSIVE, B)
MPI_Get(X) /* ok, read from public window */
MPI_Win_unlock(B)
```

**Example 11.7** In the RMA unified model, although the public and private copies of the windows are synchronized, caution must be used when combining load/stores and multi-process synchronization. Although the following example appears correct, the compiler or hardware may delay the store to X after the barrier, possibly resulting in the MPI_GET returning an incorrect value of X.

```
Process A:                Process B:
                          window location X

                          store X /* update to private & public copy of B */
MPI_Barrier               MPI_Barrier
MPI_Win_lock_all
MPI_Get(X) /* ok, read from window */
MPI_Win_flush_local(B)
/* read value in X */
MPI_Win_unlock_all
```

MPI_BARRIER provides process synchronization, but not memory synchronization. The example could potentially be made safe through the use of compiler- and hardware-specific notations to ensure the store to X occurs before process B enters the MPI_BARRIER. The use of one-sided synchronization calls, as shown in Example 11.6, also ensures the correct result.

**Example 11.8** The following example demonstrates the reading of a memory location updated by a remote process (Rule 6) in the RMA separate memory model. Although the MPI_WIN_UNLOCK on process A and the MPI_BARRIER ensure that the public copy on process B reflects the updated value of X, the call to MPI_WIN_LOCK by process B is necessary to synchronize the private copy with the public copy.

```
Process A:                    Process B:
                              window location X

MPI_Win_lock(EXCLUSIVE, B)
MPI_Put(X) /* update to public window */
```

```
MPI_Win_unlock(B)

MPI_Barrier                 MPI_Barrier

                            MPI_Win_lock(EXCLUSIVE, B)
                            /* now visible in private copy of B */
                            load X
                            MPI_Win_unlock(B)
```

Note that in this example, the barrier is not critical to the semantic correctness. The use of exclusive locks guarantees a remote process will not modify the public copy after MPI_WIN_LOCK synchronizes the private and public copies. A polling implementation looking for changes in X on process B would be semantically correct. The barrier is required to ensure that process A performs the put operation before process B performs the load of X.

**Example 11.9** Similar to Example 11.7, the following example is unsafe even in the unified model, because the load of X can not be guaranteed to occur after the MPI_BARRIER. While Process B does not need to explicitly synchronize the public and private copies through MPI_WIN_LOCK as the MPI_PUT will update both the public and private copies of the window, the scheduling of the load could result in old values of X being returned. Compiler and hardware specific notations could ensure the load occurs after the data is updated, or explicit one-sided synchronization calls can be used to ensure the proper result.

```
Process A:                  Process B:
                            window location X
MPI_Win_lock_all
MPI_Put(X) /* update to window */
MPI_Win_flush(B)

MPI_Barrier                 MPI_Barrier
                            load X
MPI_Win_unlock_all
```

**Example 11.10** The following example further clarifies Rule 5. MPI_WIN_LOCK and MPI_WIN_LOCK_ALL do *not* update the public copy of a window with changes to the private copy. Therefore, there is no guarantee that process A in the following sequence will see the value of X as updated by the local store by process B before the lock.

```
Process A:                  Process B:
                            window location X

                            store X /* update to private copy of B */
                            MPI_Win_lock(SHARED, B)
MPI_Barrier                 MPI_Barrier

MPI_Win_lock(SHARED, B)
MPI_Get(X) /* X may be the X before the store */
```

```
MPI_Win_unlock(B)
                           MPI_Win_unlock(B)
                           /* update on X now visible in public window */
```

The addition of an MPI_WIN_SYNC before the call to MPI_BARRIER by process B would guarantee process A would see the updated value of X, as the public copy of the window would be explicitly synchronized with the private copy.

**Example 11.11** Similar to the previous example, Rule 5 can have unexpected implications for general active target synchronization with the RMA separate memory model. It is *not* guaranteed that process B reads the value of X as per the local update by process A, because neither MPI_WIN_WAIT nor MPI_WIN_COMPLETE calls by process A ensure visibility in the public window copy.

```
Process A:                 Process B:
window location X
window location Y

store Y
MPI_Win_post(A, B) /* Y visible in public window */
MPI_Win_start(A)           MPI_Win_start(A)

store X /* update to private window */

MPI_Win_complete           MPI_Win_complete
MPI_Win_wait
/* update on X may not yet visible in public window */

MPI_Barrier                MPI_Barrier

                           MPI_Win_lock(EXCLUSIVE, A)
                           MPI_Get(X) /* may return an obsolete value */
                           MPI_Get(Y)
                           MPI_Win_unlock(A)
```

To allow process B to read the value of X stored by A the local store must be replaced by a local MPI_PUT that updates the public window copy. Note that by this replacement X may become visible in the private copy of process A only after the MPI_WIN_WAIT call in process A. The update to Y made before the MPI_WIN_POST call is visible in the public window after the MPI_WIN_POST call and therefore process B will read the proper value of Y. The MPI_GET(Y) call could be moved to the epoch started by the MPI_WIN_START operation, and process B would still get the value stored by process A.

**Example 11.12** The following example demonstrates the interaction of general active target synchronization with local read operations with the RMA separate memory model. Rules 5 and 6 do *not* guarantee that the private copy of X at process B has been updated before the load takes place.

```
Process A:                   Process B:
                             window location X

MPI_Win_lock(EXCLUSIVE, B)
MPI_Put(X) /* update to public window */
MPI_Win_unlock(B)

MPI_Barrier                  MPI_Barrier

                             MPI_Win_post(B)
                             MPI_Win_start(B)

                             load X /* access to private window */
                                    /* may return an obsolete value */

                             MPI_Win_complete
                             MPI_Win_wait
```

To ensure that the value put by process A is read, the local load must be replaced with a local MPI_GET operation, or must be placed after the call to MPI_WIN_WAIT.

### 11.7.1 Atomicity

The outcome of concurrent accumulate operations to the same location with the same predefined datatype is as if the accumulates were done at that location in some serial order. Additional restrictions on the operation apply; see the info key accumulate_ops in Section 11.2.1. Concurrent accumulate operations with different origin and target pairs are not ordered. Thus, there is no guarantee that the entire call to an accumulate operation is executed atomically. The effect of this lack of atomicity is limited: The previous correctness conditions imply that a location updated by a call to an accumulate operation cannot be accessed by a load or an RMA call other than accumulate until the accumulate operation 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. The outcome of accumulate operations with overlapping types of different sizes or target displacements is undefined.

### 11.7.2 Ordering

Accumulate calls enable element-wise atomic read and write to remote memory locations. MPI specifies ordering between accumulate operations from one process to the same (or overlapping) memory locations at another process on a per-datatype granularity. The default ordering is strict ordering, which guarantees that overlapping updates from the same source to a remote location are committed in program order and that reads (e.g., with MPI_GET_ACCUMULATE) and writes (e.g., with MPI_ACCUMULATE) are executed and committed in program order. Ordering only applies to operations originating at the same origin that access overlapping target memory regions. MPI does not provide any guarantees for accesses or updates from different origin processes to overlapping target memory regions.

The default strict ordering may incur a significant performance penalty. MPI specifies the info key accumulate_ordering to allow relaxation of the ordering semantics when specified to any window creation function. The values for this key are as follows. If set to none, then no ordering will be guaranteed for accumulate calls. This was the behavior for RMA in MPI-2 but is *not* the default in MPI-3. The key can be set to a comma-separated list of required access orderings at the target. Allowed values in the comma-separated list are rar, war, raw, and waw for read-after-read, write-after-read, read-after-write, and write-after-write ordering, respectively. These indicate whether operations of the specified type complete in the order they were issued. For example, raw means that any writes must complete at the target before subsequent reads. These ordering requirements apply only to operations issued by the same origin process and targeting the same target process. The default value for accumulate_ordering is rar,raw,war,waw, which implies that writes complete at the target in the order in which they were issued, reads complete at the target before any writes that are issued after the reads, and writes complete at the target before any reads that are issued after the writes. Any subset of these four orderings can be specified. For example, if only read-after-read and write-after-write ordering is required, then the value of the accumulate_ordering key could be set to rar,waw. The order of values is not significant.

Note that the above ordering semantics apply only to accumulate operations, not put and get. Put and get within an epoch are unordered.

### 11.7.3 Progress

One-sided communication has the same progress requirements as point-to-point communication: once a communication is enabled it is guaranteed to complete. RMA calls must have local semantics, except when required for synchronization with other RMA calls.

There is some fuzziness in the definition of the time when a RMA communication becomes enabled. This fuzziness provides to the implementor more flexibility than with point-to-point communication. Access to a target window becomes enabled once the corresponding synchronization (such as MPI_WIN_FENCE or MPI_WIN_POST) has executed. On the origin process, an RMA communication may become enabled as soon as the corresponding put, get or accumulate call has executed, or as late as when the ensuing synchronization call is issued. Once the communication is enabled both at the origin and at the target, the communication must complete.

Consider the code fragment in Example 11.4. Some of the calls may block if the target window is not posted. However, if the target window is posted, then the code fragment must complete. The data transfer may start as soon as the put call occurs, but may be delayed until the ensuing complete call occurs.

Consider the code fragment in Example 11.5. Some of the calls may block if another process holds a conflicting lock. However, if no conflicting lock is held, then the code fragment must complete.

Consider the code illustrated in Figure 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.

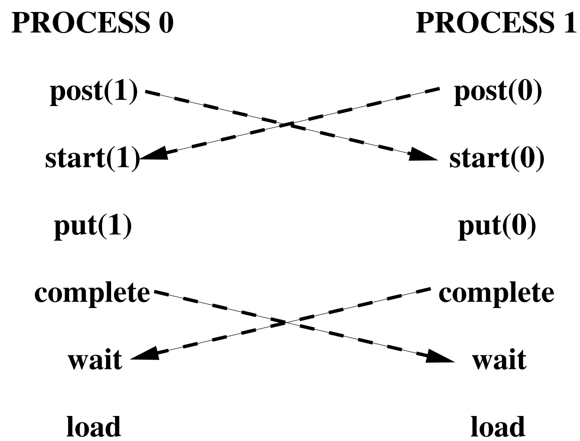


Figure 11.6: Symmetric communication

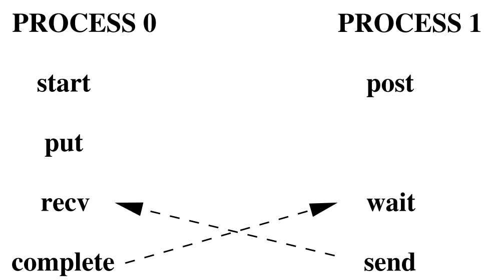


Figure 11.7: Deadlock situation

The following two examples illustrate the fact that the synchronization between complete and wait is not symmetric: the wait call blocks until the complete executes, but not vice versa. Consider the code illustrated in Figure 11.7. This code will deadlock: the wait of process 1 blocks until process 0 calls complete, and the receive of process 0 blocks until process 1 calls send. Consider, on the other hand, the code illustrated in Figure 11.8. This code will not deadlock. Once process 1 calls post, then the sequence start, put, complete on process 0 can proceed to completion. Process 0 will reach the send call, allowing the receive call of process 1 to complete.

*Rationale.* MPI implementations must guarantee that a process makes progress on all enabled communications it participates in, while blocked on an MPI call. This is true

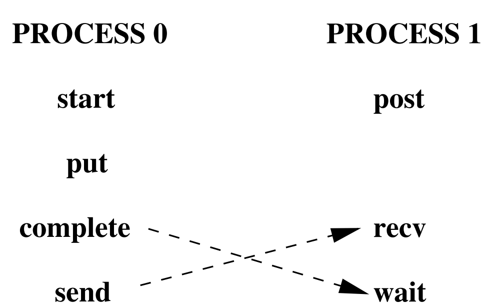


Figure 11.8: No deadlock

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, the MPI Forum decided not to define which interpretation of the standard is the correct one, since the issue is contentious. (*End of rationale.*)

### 11.7.4 Registers and Compiler Optimizations

*Advice to users.* All the material in this section is an advice to users. (*End of advice to users.*)

A coherence problem exists between variables kept in registers and the memory values of these variables. An RMA call may access a variable in memory (or cache), while the up-to-date value of this variable is in register. A get will not return the latest variable value, and a put may be overwritten when the register is stored back in memory. Note that these issues are unrelated to the RMA memory model; that is, these issues apply even if the memory model is MPI_WIN_UNIFIED.

The problem is illustrated by the following code:

```
Source of Process 1     Source of Process 2    Executed in Process 2
bbbb = 777              buff = 999             reg_A:=999
call MPI_WIN_FENCE      call MPI_WIN_FENCE
call MPI_PUT(bbbb                              stop appl. thread
into buff of process 2)                        buff:=777 in PUT handler
                                               continue appl. thread
call MPI_WIN_FENCE      call MPI_WIN_FENCE
                        ccc = buff             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 afflicts in some cases send/receive communication, is discussed more at length in Section 17.1.16.

Programs written in C avoid this problem, because of the semantics of C. Many Fortran compilers will avoid this problem, without disabling compiler optimizations. However, in order to avoid register coherence problems in a completely portable manner, users should restrict their use of RMA windows to variables stored in modules or COMMON blocks. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20. Sections 17.1.17 to 17.1.17 discuss several solutions for the problem in this example.

## 11.8 Examples

**Example 11.13** The following example shows a generic loosely synchronous, iterative code, using fence synchronization. The window at each process consists of array A, which contains the origin and target buffers of the put calls.

```
...
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 11.14** Same generic example, with more computation/communication overlap. We assume that the update phase is broken into two subphases: the first, where the "boundary," which is involved in communication, is updated, and the second, where the "core," which neither uses nor provides communicated data, is updated.

```
...
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],
                    fromdisp[i], 1, fromtype[i], win);
  update_core(A);
  MPI_Win_fence(MPI_MODE_NOSUCCEED, 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 11.15** Same code as in Example 11.13, rewritten using post-start-complete-wait.

```
...
while (!converged(A)) {
  update(A);
  MPI_Win_post(fromgroup, 0, win);
  MPI_Win_start(togroup, 0, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                todisp[i], 1, totype[i], win);
  MPI_Win_complete(win);
  MPI_Win_wait(win);
}
```

**Example 11.16** Same example, with split phases, as in Example 11.14.

```
...
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 11.17** A checkerboard, or double buffer communication pattern, that allows more computation/communication overlap. Array `A0` is updated using values of array `A1`, and vice versa. We assume that communication is symmetric: if process A gets data from process B, then process B gets data from process A. Window `wini` consists of array `Ai`.

```
...
if (!converged(A0,A1))
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win0);
MPI_Barrier(comm0);
/* the barrier is needed because the start call inside the
loop uses the nocheck option */
while (!converged(A0, A1)) {
  /* communication on A0 and computation on A1 */
  update2(A1, A0); /* local update of A1 that depends on A0 (and A1) */
  MPI_Win_start(neighbors, MPI_MODE_NOCHECK, win0);
  for(i=0; i < fromneighbors; i++)
    MPI_Get(&tobuf0[i], 1, totype0[i], neighbor[i],
                   fromdisp0[i], 1, fromtype0[i], win0);
  update1(A1); /* local update of A1 that is
```

```
                concurrent with communication that updates A0 */
  MPI_Win_post(neighbors, (MPI_MODE_NOCHECK | MPI_MODE_NOPUT), win1);
  MPI_Win_complete(win0);
  MPI_Win_wait(win0);

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

A process posts the local window associated with win0 before it completes RMA accesses to the remote windows associated with win1. When the wait(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.

In the next several examples, for conciseness, the expression

```
z = MPI_Get_accumulate(...)
```

means to perform an MPI_GET_ACCUMULATE with the result buffer (given by result_addr in the description of MPI_GET_ACCUMULATE) on the left side of the assignment, in this case, z. This format is also used with MPI_COMPARE_AND_SWAP.

**Example 11.18** The following example implements a naive, non-scalable counting semaphore. The example demonstrates the use of MPI_WIN_SYNC to manipulate the public copy of X, as well as MPI_WIN_FLUSH to complete operations without ending the access epoch opened with MPI_WIN_LOCK_ALL. To avoid the rules regarding synchronization of the public and private copies of windows, MPI_ACCUMULATE and MPI_GET_ACCUMULATE are used to write to or read from the local public copy.

```
Process A:                               Process B:
MPI_Win_lock_all                         MPI_Win_lock_all
window location X
X=2
MPI_Win_sync
MPI_Barrier                              MPI_Barrier
```

```
MPI_Accumulate(X, MPI_SUM, -1)           MPI_Accumulate(X, MPI_SUM, -1)

stack variable z                         stack variable z
do                                       do
  z = MPI_Get_accumulate(X,                z = MPI_Get_accumulate(X,
       MPI_NO_OP, 0)                            MPI_NO_OP, 0)
  MPI_Win_flush(A)                         MPI_Win_flush(A)
while(z!=0)                              while(z!=0)

MPI_Win_unlock_all                       MPI_Win_unlock_all
```

**Example 11.19** Implementing a critical region between two processes (Peterson's algorithm). Despite their appearance in the following example, MPI_WIN_LOCK_ALL and MPI_WIN_UNLOCK_ALL are not collective calls, but it is frequently useful to start shared access epochs to all processes from all other processes in a window. Once the access epochs are established, accumulate communication operations and flush and sync synchronization operations can be used to read from or write to the public copy of the window.

```
Process A:                           Process B:
window location X                    window location Y
window location T

MPI_Win_lock_all                     MPI_Win_lock_all
X=1                                  Y=1
MPI_Win_sync                         MPI_Win_sync
MPI_Barrier                          MPI_Barrier
MPI_Accumulate(T, MPI_REPLACE, 1)    MPI_Accumulate(T, MPI_REPLACE, 0)
stack variables t,y                  stack variable t,x
t=1                                  t=0
y=MPI_Get_accumulate(Y,              x=MPI_Get_accumulate(X,
   MPI_NO_OP, 0)                        MPI_NO_OP, 0)
while(y==1 && t==1) do               while(x==1 && t==0) do
  y=MPI_Get_accumulate(Y,              x=MPI_Get_accumulate(X,
     MPI_NO_OP, 0)                        MPI_NO_OP, 0)
  t=MPI_Get_accumulate(T,              t=MPI_Get_accumulate(T,
     MPI_NO_OP, 0)                        MPI_NO_OP, 0)
  MPI_Win_flush_all                    MPI_Win_flush(A)
done                                 done
// critical region                   // critical region
MPI_Accumulate(X, MPI_REPLACE, 0)    MPI_Accumulate(Y, MPI_REPLACE, 0)
MPI_Win_unlock_all                   MPI_Win_unlock_all
```

**Example 11.20** Implementing a critical region between multiple processes with compare and swap. The call to MPI_WIN_SYNC is necessary on Process A after local initialization of `A` to guarantee the public copy has been updated with the initialization value found in the private copy. It would also be valid to call MPI_ACCUMULATE with MPI_REPLACE to directly initialize the public copy. A call to MPI_WIN_FLUSH would be necessary to assure A in the public copy of Process A had been updated before the barrier.

```
Process A:                          Process B...:
MPI_Win_lock_all                    MPI_Win_lock_all
atomic location A
A=0
MPI_Win_sync
MPI_Barrier                         MPI_Barrier
stack variable r=1                  stack variable r=1
while(r != 0) do                    while(r != 0) do
  r = MPI_Compare_and_swap(A, 0, 1)   r = MPI_Compare_and_swap(A, 0, 1)
  MPI_Win_flush(A)                    MPI_Win_flush(A)
done                                done
// critical region                  // critical region
r = MPI_Compare_and_swap(A, 1, 0)   r = MPI_Compare_and_swap(A, 1, 0)
MPI_Win_unlock_all                  MPI_Win_unlock_all
```

**Example 11.21** The following example demonstrates the proper synchronization in the unified memory model when a data transfer is implemented with load and store in the case of windows in shared memory (instead of MPI_PUT or MPI_GET) and the synchronization between processes is performed using point-to-point communication. The synchronization between processes must be supplemented with a memory synchronization through calls to MPI_WIN_SYNC, which act locally as a processor-memory barrier. In Fortran, if MPI_ASYNC_PROTECTS_NONBLOCKING is .FALSE. or the variable X is not declared as ASYNCHRONOUS, reordering of the accesses to the variable X must be prevented with MPI_F_SYNC_REG operations. (No equivalent function is needed in C.)

The variable X is contained within a shared memory window and X corresponds to the same memory location at both processes. The MPI_WIN_SYNC operation performed by process A ensures completion of the load/store operations issued by process A. The MPI_WIN_SYNC operation performed by process B ensures that process A's updates to X are visible to process B.

```
Process A                        Process B

MPI_WIN_LOCK_ALL(                MPI_WIN_LOCK_ALL(
      MPI_MODE_NOCHECK,win)            MPI_MODE_NOCHECK,win)

DO ...                           DO ...
  X=...

  MPI_F_SYNC_REG(X)
  MPI_WIN_SYNC(win)
  MPI_SEND                         MPI_RECV
                                   MPI_WIN_SYNC(win)
                                   MPI_F_SYNC_REG(X)

                                   print X
```

```
                                    MPI_F_SYNC_REG(X)
  MPI_RECV                          MPI_SEND
  MPI_F_SYNC_REG(X)
END DO                            END DO

MPI_WIN_UNLOCK_ALL(win)           MPI_WIN_UNLOCK_ALL(win)
```

**Example 11.22** The following example shows how request-based operations can be used to overlap communication with computation. Each process fetches, processes, and writes the result for NSTEPS chunks of data. Instead of a single buffer, M local buffers are used to allow up to M communication operations to overlap with computation.

```
int         i, j;
MPI_Win     win;
MPI_Request put_req[M] = { MPI_REQUEST_NULL };
MPI_Request get_req;
double      *baseptr;
double      data[M][N];

MPI_Win_allocate(NSTEPS*N*sizeof(double), sizeof(double), MPI_INFO_NULL,
  MPI_COMM_WORLD, &baseptr, &win);

MPI_Win_lock_all(0, win);

for (i = 0; i < NSTEPS; i++) {
 if (i<M)
   j=i;
 else
   MPI_Waitany(M, put_req, &j, MPI_STATUS_IGNORE);

 MPI_Rget(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
          &get_req);
 MPI_Wait(&get_req,MPI_STATUS_IGNORE);
 compute(i, data[j], ...);
 MPI_Rput(data[j], N, MPI_DOUBLE, target, i*N, N, MPI_DOUBLE, win,
          &put_req[j]);
}

MPI_Waitall(M, put_req, MPI_STATUSES_IGNORE);
MPI_Win_unlock_all(win);
```

**Example 11.23** The following example constructs a distributed shared linked list using dynamic windows. Initially process 0 creates the head of the list, attaches it to the window, and broadcasts the pointer to all processes. All processes then concurrently append N new elements to the list. When a process attempts to attach its element to the tail of the list it may discover that its tail pointer is stale and it must chase ahead to the new tail before the element can be attached. This example requires some modification to work in an environment where the layout of the structures is different on different processes.

```
...
#define NUM_ELEMS 10

#define LLIST_ELEM_NEXT_RANK ( offsetof(llist_elem_t, next) + \
                               offsetof(llist_ptr_t, rank) )
#define LLIST_ELEM_NEXT_DISP ( offsetof(llist_elem_t, next) + \
                               offsetof(llist_ptr_t, disp) )

/* Linked list pointer */
typedef struct {
  MPI_Aint disp;
  int      rank;
} llist_ptr_t;

/* Linked list element */
typedef struct {
  llist_ptr_t next;
  int value;
} llist_elem_t;

const llist_ptr_t nil = { (MPI_Aint) MPI_BOTTOM, -1 };

/* List of locally allocated list elements. */
static llist_elem_t **my_elems = NULL;
static int my_elems_size  = 0;
static int my_elems_count = 0;

/* Allocate a new shared linked list element */
MPI_Aint alloc_elem(int value, MPI_Win win) {
  MPI_Aint disp;
  llist_elem_t *elem_ptr;

  /* Allocate the new element and register it with the window */
  MPI_Alloc_mem(sizeof(llist_elem_t), MPI_INFO_NULL, &elem_ptr);
  elem_ptr->value = value;
  elem_ptr->next  = nil;
  MPI_Win_attach(win, elem_ptr, sizeof(llist_elem_t));

  /* Add the element to the list of local elements so we can free
     it later. */
  if (my_elems_size == my_elems_count) {
    my_elems_size += 100;
    my_elems = realloc(my_elems, my_elems_size*sizeof(void*));
  }
  my_elems[my_elems_count] = elem_ptr;
```

```
  my_elems_count++;

  MPI_Get_address(elem_ptr, &disp);
  return disp;
}

int main(int argc, char *argv[]) {
  int           procid, nproc, i;
  MPI_Win       llist_win;
  llist_ptr_t   head_ptr, tail_ptr;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &procid);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &llist_win);

  /* Process 0 creates the head node */
  if (procid == 0)
    head_ptr.disp = alloc_elem(-1, llist_win);

  /* Broadcast the head pointer to everyone */
  head_ptr.rank = 0;
  MPI_Bcast(&head_ptr.disp, 1, MPI_AINT, 0, MPI_COMM_WORLD);
  tail_ptr = head_ptr;

  /* Lock the window for shared access to all targets */
  MPI_Win_lock_all(0, llist_win);

  /* All processes concurrently append NUM_ELEMS elements to the list */
  for (i = 0; i < NUM_ELEMS; i++) {
    llist_ptr_t new_elem_ptr;
    int success;

    /* Create a new list element and attach it to the window */
    new_elem_ptr.rank = procid;
    new_elem_ptr.disp = alloc_elem(procid, llist_win);

    /* Append the new node to the list.  This might take multiple
       attempts if others have already appended and our tail pointer
       is stale. */
    do {
      llist_ptr_t next_tail_ptr = nil;

      MPI_Compare_and_swap((void*) &new_elem_ptr.rank, (void*) &nil.rank,
          (void*)&next_tail_ptr.rank, MPI_INT, tail_ptr.rank,
          MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_RANK),
```

```
        llist_win);

      MPI_Win_flush(tail_ptr.rank, llist_win);
      success = (next_tail_ptr.rank == nil.rank);

      if (success) {
        MPI_Accumulate(&new_elem_ptr.disp, 1, MPI_AINT, tail_ptr.rank,
            MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP), 1,
            MPI_AINT, MPI_REPLACE, llist_win);

        MPI_Win_flush(tail_ptr.rank, llist_win);
        tail_ptr = new_elem_ptr;

      } else {
        /* Tail pointer is stale, fetch the displacement.  May take
           multiple tries if it is being updated. */
        do {
          MPI_Get_accumulate(NULL, 0, MPI_AINT, &next_tail_ptr.disp,
              1, MPI_AINT, tail_ptr.rank,
              MPI_Aint_add(tail_ptr.disp, LLIST_ELEM_NEXT_DISP),
              1, MPI_AINT, MPI_NO_OP, llist_win);

          MPI_Win_flush(tail_ptr.rank, llist_win);
        } while (next_tail_ptr.disp == nil.disp);
        tail_ptr = next_tail_ptr;
      }
    } while (!success);
  }

  MPI_Win_unlock_all(llist_win);
  MPI_Barrier(MPI_COMM_WORLD);

  /* Free all the elements in the list */
  for ( ; my_elems_count > 0; my_elems_count--) {
    MPI_Win_detach(llist_win,my_elems[my_elems_count-1]);
    MPI_Free_mem(my_elems[my_elems_count-1]);
  }
  MPI_Win_free(&llist_win);
...
```

# 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. These calls can be used to layer new functionality on top of MPI. Next, Section 12.3 deals with setting the information found in status. This functionality 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 to replace these mechanisms: it is assumed programmers who wish to define new asynchronous operations will use the mechanisms provided by the underlying operating system. Thus, the calls in this section only provide a means for defining the effect of MPI calls such as MPI_WAIT or MPI_CANCEL when they apply to generalized requests, and for signaling to MPI the completion of a generalized operation.

> *Rationale.* It is tempting to also define an MPI standard mechanism for achieving concurrent execution of user-defined nonblocking operations. However, it is difficult to define such a mechanism without consideration of the specific mechanisms used in the operating system. The Forum feels that concurrency mechanisms are a proper part of the underlying operating system and should not be standardized by MPI; the MPI standard should only deal with the interaction of such mechanisms with MPI. (*End of rationale.*)

For a regular request, the operation associated with the request is performed by the MPI implementation, and the operation completes without intervention by the application. For a generalized request, the operation associated with the request is performed by the application; therefore, the application must notify MPI through a call to MPI_GREQUEST_COMPLETE when the operation completes. 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

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

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

**C binding**

```
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)
```

**F08 binding**

```
MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
             ierror)
    PROCEDURE(MPI_Grequest_query_function) ::  query_fn
    PROCEDURE(MPI_Grequest_free_function) ::  free_fn
    PROCEDURE(MPI_Grequest_cancel_function) ::  cancel_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_GREQUEST_START(QUERY_FN, FREE_FN, CANCEL_FN, EXTRA_STATE, REQUEST,
             IERROR)
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER REQUEST, IERROR
```

*Advice to users.* Note that a generalized request 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; extra_state can be used to maintain user-defined state for the request.

In C, the query function is

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

in Fortran with the mpi_f08 module

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
      TYPE(MPI_Status) ::  status
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state
      INTEGER ::  ierror
```

in Fortran with the mpi module and mpif.h

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

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

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

In C, the free function is

```
typedef int MPI_Grequest_free_function(void *extra_state);
```

in Fortran with the mpi_f08 module

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state
      INTEGER ::  ierror
```

in Fortran with the mpi module and mpif.h

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

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

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

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

*Advice to users.* Calling MPI_REQUEST_FREE(request) will cause the request handle 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 in which 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 with the mpi_f08 module

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state
      LOGICAL ::  complete
      INTEGER ::  ierror
```

in Fortran with the mpi module and mpif.h

```
SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
    INTEGER IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    LOGICAL COMPLETE
```

The cancel_fn function is invoked to start the cancelation of a generalized request. It is called by MPI_CANCEL(request). MPI passes complete=true to the callback function if MPI_GREQUEST_COMPLETE was already called on the request, and complete=false otherwise.

All callback functions return an error code. The code is passed back and dealt with as appropriate for the error code by the MPI function that invoked the callback function. For example, if error codes are returned then the error code returned by the callback function will be returned by the MPI function that invoked the callback function. In the case of an MPI_{WAIT|TEST}{ANY} call that invokes both query_fn and free_fn, the MPI call will return the error code returned by the last callback, namely free_fn. If one or more of the requests in a call to MPI_{WAIT|TEST}{SOME|ALL} failed, then the MPI call will return MPI_ERR_IN_STATUS. In such a case, if the MPI call was passed an array of statuses, then MPI will return in each of the statuses that correspond to a completed generalized request the error code returned by the corresponding invocation of its free_fn callback function. However, if the MPI function was passed MPI_STATUSES_IGNORE, then the individual error codes returned by each callback functions will be lost.

*Advice to users.* query_fn must *not* set the error field of status since query_fn may be called by MPI_WAIT or MPI_TEST, in which case the error field of status should not change. The MPI library knows the "context" in which query_fn is invoked and can decide correctly when to put the returned error code in the error field of status. (*End of advice to users.*)

MPI_GREQUEST_COMPLETE(request)

| | | |
|---|---|---|
| INOUT | request | generalized request (handle) |

**C binding**

```
int MPI_Grequest_complete(MPI_Request request)
```

**F08 binding**

```
MPI_Grequest_complete(request, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_GREQUEST_COMPLETE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

The call informs MPI that the operations represented by the generalized request request are complete (see definitions in Section 2.4). A call to MPI_WAIT(request, status) will return and a call to MPI_TEST(request, flag, status) will return flag=true only after a call to MPI_GREQUEST_COMPLETE has declared that these operations are complete.

MPI imposes no restrictions on the code executed by the callback functions. However, new nonblocking operations should be defined so that the general semantic rules about MPI calls such as MPI_TEST, MPI_REQUEST_FREE, or MPI_CANCEL still hold. For example, these calls are supposed to be local and nonblocking. Therefore, the callback functions query_fn, free_fn, or cancel_fn should invoke blocking MPI communication calls only if the context is such that these calls are guaranteed to return in finite time. Once MPI_CANCEL is invoked, the cancelled operation should complete in finite time, irrespective of the state of other processes (the operation has acquired "local" semantics). It should either succeed, or fail without side-effects. The user should guarantee these same properties for newly defined operations.

*Advice to implementors.* A call to MPI_GREQUEST_COMPLETE may unblock a blocked user process/thread. The MPI library should ensure that the blocked user computation will resume. (*End of advice to implementors.*)

### 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;


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 */
   /* The availability of the pthread_create call is system dependent */
   pthread_create(&thread, NULL, reduce_thread, args);

   return MPI_SUCCESS;
}

/* thread code */
void* reduce_thread(void *ptr)
{
   int lchild, rchild, parent, lval, rval, val;
```

```
   MPI_Request req[2];
   ARGS *args;

   args = (ARGS*)ptr;

   /* compute left and 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 */
   MPI_Status_set_cancelled(status, 0);
   /* 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");
      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 to use the same request mechanism, which 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 values for a given request are defined in the sections with the new request.

Generalized requests raise additional considerations. Here, the user provides the functions to deal with the request. Unlike other MPI calls, the user needs to provide the information to be returned in the status. The status argument is provided directly to the callback function where the status needs to be set. Users can directly set the values in 3 of the 5 status values. The count and cancel fields are opaque. To overcome this, these calls are provided:

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 (non-negative integer) |

**C binding**

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

**F08 binding**

```
MPI_Status_set_elements(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(INOUT) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_STATUS_SET_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR
```

MPI_STATUS_SET_ELEMENTS_X(status, datatype, count)

| | | |
|---|---|---|
| INOUT | status | status with which to associate count (Status) |
| IN | datatype | datatype associated with count (handle) |
| IN | count | number of elements to associate with status (integer) |

**C binding**

```
int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
             MPI_Count count)
```

**F08 binding**

```
MPI_Status_set_elements_x(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(INOUT) :: status
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) :: count
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) COUNT
```

These functions modify the opaque part of status so that a call to MPI_GET_ELEMENTS or MPI_GET_ELEMENTS_X will return count. MPI_GET_COUNT will return a compatible value.

> *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), MPI_GET_ELEMENTS(status, datatype, count), or MPI_GET_ELEMENTS_X(status, datatype, count) must use a datatype argument that has the same type signature as the datatype argument that was used in the call to MPI_STATUS_SET_ELEMENTS or MPI_STATUS_SET_ELEMENTS_X.

> *Rationale.* The requirement of matching type signatures for these calls is similar to the restriction that holds when count is set by a receive operation: in that case, the calls to MPI_GET_COUNT, MPI_GET_ELEMENTS, and MPI_GET_ELEMENTS_X must use a datatype with the same signature as the datatype used in the receive call. (*End of rationale.*)

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 binding**

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

**F08 binding**

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

**F binding**

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

If flag is set to true then a subsequent call to MPI_TEST_CANCELLED(status, flag) will also return flag = true, otherwise it will return false.

> *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, MPI implementations are not required to be thread compliant as defined in this section. Regardless of whether or not the MPI implementation is thread compliant, MPI_INITIALIZED, MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION and MPI_GET_LIBRARY_VERSION must always be thread-safe. When a thread is executing one of these routines, if another concurrently running thread also makes an MPI call, the outcome will be as if the calls executed in some order.

This section generally assumes a thread package similar to POSIX threads [39], 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 in which MPI 'processes' are POSIX threads inside a single POSIX process are not thread-compliant by this definition (indeed, their "processes" are single-threaded). (*End of rationale.*)

*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 receive call will occur. Once both calls occur, the communication is enabled and both calls will complete. On the other hand, a single-threaded process that posts a send, followed by a matching receive, may deadlock. The progress requirement for multithreaded implementations is stronger, as a blocked call cannot prevent progress in other threads.

*Advice to implementors.* MPI calls can be made thread-safe by executing only one at a time, e.g., by protecting MPI code with one process-global lock. However, blocked operations cannot hold the lock, as this would prevent progress of other threads in the process. The lock is held only for the duration of an atomic, locally-completing suboperation such as posting a send or completing a send, and is released in between. Finer locks can provide more concurrency, at the expense of higher locking overheads. Concurrency can also be achieved by having some of the MPI protocol executed by separate server threads. (*End of advice to implementors.*)

### 12.4.2 Clarifications

Initialization and Completion The call to MPI_FINALIZE should occur on the same thread that initialized MPI. We call this thread the **main thread**. The call should occur only after all 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 in which two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent MPI_{WAIT|TEST}{ANY|SOME|ALL} calls. In MPI, a request can only be completed once. Any combination of wait or test that violates this rule is erroneous.

*Rationale.* This restriction is consistent with the view that a multithreaded execution corresponds to an interleaving of the MPI calls. In a single threaded implementation, once a wait is posted on a request the request handle will be nullified before it is possible to post a second wait on the same handle. With threads, an MPI_WAIT{ANY|SOME|ALL} may be blocked without having nullified its request(s) so it becomes the user's responsibility to avoid using the same request in an MPI_WAIT on another thread. This constraint also simplifies implementation, as only one thread will be blocked on any communication or I/O event. (*End of rationale.*)

Probe A receive call that uses source and tag values returned by a preceding call to MPI_PROBE or MPI_IPROBE will receive the message matched by the probe call only if there was no other matching receive after the probe and before that receive. In a multithreaded environment, it is up to the user to enforce this condition using suitable mutual exclusion logic. This can be enforced by making sure that each communicator is used by only one thread on each process. Alternatively, MPI_MPROBE or MPI_IMPROBE can be used.

Collective calls Matching of collective calls on a communicator, window, or file handle is done according to the order in which the calls are issued at each process. If concurrent 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 specified in MPI_FILE_OPEN and MPI_WIN_CREATE, a file handle and a window handle inherit only the group of processes of the underlying communicator, but not the communicator itself. Accesses to communicators, window handles and file handles cannot affect one another. (*End of rationale.*)

*Advice to implementors.* If the implementation of file or window operations internally uses MPI communication then a duplicated communicator may be cached on the file or window object. (*End of advice to implementors.*)

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 at which the thread executing them may be cancelled. The above restriction simplifies implementation (no need for the MPI library to be "async-cancel-safe" or "async-signal-safe"). (*End of rationale.*)

*Advice to users.* Users can catch signals in separate, non-MPI threads (e.g., by masking signals on MPI calling threads, and unmasking them in one or more non-MPI threads). A good programming practice is to have a distinct thread blocked in a call to `sigwait` for each user expected signal that may occur. Users must not catch signals used by the MPI implementation; as each MPI implementation is required to document the signals used internally, users can avoid these signals. (*End of advice to users.*)

*Advice to implementors.* The MPI library should not invoke library calls that are not thread safe, if multiple threads execute. (*End of advice to implementors.*)

### 12.4.3 Initialization

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

MPI_INIT_THREAD(required, provided)

| | | |
|---|---|---|
| IN | required | desired level of thread support (integer) |
| OUT | provided | provided level of thread support (integer) |

**C binding**

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

**F08 binding**

```
MPI_Init_thread(required, provided, ierror)
    INTEGER, INTENT(IN) :: required
    INTEGER, INTENT(OUT) :: provided
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
    INTEGER REQUIRED, PROVIDED, IERROR
```

> *Advice to users.* In C, the passing of argc and argv is optional, as with MPI_INIT as discussed in Section 8.7. In C, null pointers may be passed in their place. (*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 **??**).

**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.

An MPI library that is not thread compliant must always return provided=MPI_THREAD_SINGLE, even if MPI_INIT_THREAD is called on a multithreaded process. The library should also return correct values for the MPI calls that can be executed before initialization, even if multiple threads have been spawned.

> *Rationale.* Such code is erroneous, but if the MPI initialization is performed by a library, the error cannot be detected until MPI_INIT_THREAD is called. The requirements in the previous paragraph ensure that the error can be properly detected. (*End of rationale.*)

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, instead, 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; alternatively, 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 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 or Fortran library calls that are not thread safe, e.g., in an environment where `malloc` is not thread safe, then `malloc` should not be used by the MPI library.
>
> Some implementors may want to use different MPI libraries for different levels of thread support. They can do so using dynamic linking and selecting which library will be linked when MPI_INIT_THREAD is invoked. If this is not possible, then optimizations for lower levels of thread support will occur only when the level of thread support required is specified at link time.
>
> Note that required need not be the same value on all processes of MPI_COMM_WORLD. (*End of advice to implementors.*)

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

MPI_QUERY_THREAD(provided)

| | | |
|---|---|---|
| OUT | provided | provided level of thread support (integer) |

**C binding**

```
int MPI_Query_thread(int *provided)
```

**F08 binding**

```
MPI_Query_thread(provided, ierror)
    INTEGER, INTENT(OUT) :: provided
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR
```

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

MPI_IS_THREAD_MAIN(flag)

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

**C binding**

```
int MPI_Is_thread_main(int *flag)
```

**F08 binding**

```
MPI_Is_thread_main(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_IS_THREAD_MAIN(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR
```

This function can be called by a thread to determine if 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 can 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_GET_VERSION, MPI_INITIALIZED, or MPI_FINALIZED 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 [47], collective buffering [7, 15, 48, 51, 57], and disk-directed I/O [43]) 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 non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term “etype” is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type.

**filetype** A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, but they must be non-negative and monotonically nondecreasing.

**view** A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Figure 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).

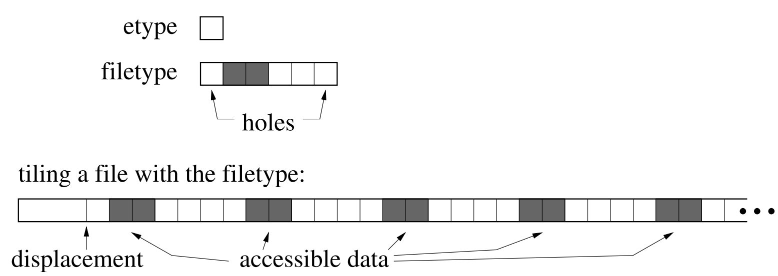


Figure 13.1: Etypes and filetypes

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

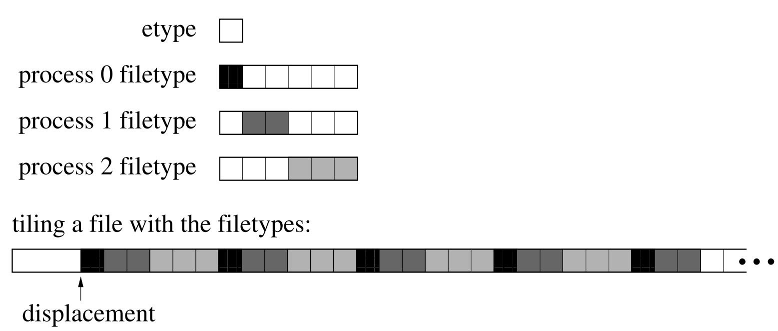


Figure 13.2: Partitioning a file among parallel processes

**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 eighth etype in the file after the displacement. An "explicit offset" is an offset that is used as an argument in explicit data access routines.

**file size and end of file** The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first etype accessible in the current view starting after the last byte in the file.

**file pointer** A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file.

**file handle** A *file handle* is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle.

## 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) |

**C binding**

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

**F08 binding**

```
MPI_File_open(comm, filename, amode, info, fh, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=*), INTENT(IN) :: filename
    INTEGER, INTENT(IN) :: amode
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_File), INTENT(OUT) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_OPEN(COMM, FILENAME, AMODE, INFO, FH, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER COMM, AMODE, INFO, FH, IERROR
```

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

The format for specifying the file name in the filename argument is implementation dependent and must be documented by the implementation.

*Advice to implementors.* An implementation may require that filename include a string or strings specifying additional information about the file. Examples include the type of filesystem (e.g., a prefix of `ufs:`), a remote hostname (e.g., a prefix of `machine.univ.edu:`), or a file password (e.g., a suffix of `/PASSWORD=SECRET`). (*End of advice to implementors.*)

*Advice to users.* On some implementations of MPI, the file namespace may not be identical from all processes of all applications. For example, "`/tmp/foo`" may denote different files on different processes, or a single file may have many names, dependent on process location. The user is responsible for ensuring that a single file is referenced by the filename argument, as it may be impossible for an implementation to detect this type of namespace error. (*End of advice to users.*)

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation (no data representation conversion is performed). (POSIX files are linear byte streams in the native representation.) The file view can be changed via the MPI_FILE_SET_VIEW routine.

The following access modes are supported (specified in amode, a bit vector OR of the following integer constants):

- MPI_MODE_RDONLY — read only,
- MPI_MODE_RDWR — reading and writing,
- MPI_MODE_WRONLY — write only,
- MPI_MODE_CREATE — create the file if it does not exist,
- MPI_MODE_EXCL — error if creating file that already exists,
- MPI_MODE_DELETE_ON_CLOSE — delete file on close,
- MPI_MODE_UNIQUE_OPEN — file will not be concurrently opened elsewhere,
- MPI_MODE_SEQUENTIAL — file will only be accessed sequentially,
- MPI_MODE_APPEND — set initial position of all file pointers to end of file.

*Advice to users.* C users can use bit vector OR (|) to combine these constants; Fortran 90 users can use the bit vector IOR intrinsic. Fortran 77 users can use (nonportably) bit vector IOR on systems that support it. Alternatively, Fortran users can portably use integer addition to OR the constants (each constant should appear at most once in the addition.). (*End of advice to users.*)

*Advice to implementors.* The values of these constants must be defined such that the bitwise OR and the sum of any distinct set of these constants is equivalent. (*End of advice to implementors.*)

The modes MPI_MODE_RDONLY, MPI_MODE_RDWR, MPI_MODE_WRONLY, MPI_MODE_CREATE, and MPI_MODE_EXCL have identical semantics to their POSIX counterparts [39]. 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). 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). 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) |

**C binding**

```
int MPI_File_close(MPI_File *fh)
```

**F08 binding**

```
MPI_File_close(fh, ierror)
    TYPE(MPI_File), INTENT(INOUT) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_CLOSE(FH, IERROR)
    INTEGER FH, IERROR
```

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 binding**

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

**F08 binding**

```
MPI_File_delete(filename, info, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: filename
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

MPI_FILE_DELETE deletes the file identified by the file name filename. If the file does not exist, MPI_FILE_DELETE raises an error in the class MPI_ERR_NO_SUCH_FILE.

The info argument can be used to provide information regarding file system specifics (see Section 13.2.8). The constant MPI_INFO_NULL refers to the null info, and can be used when no info needs to be specified.

If a process currently has the file open, the behavior of any access to the file (as well as the behavior of any outstanding accesses) is implementation dependent. In addition, whether an open file is deleted or not is also implementation dependent. If the file is not deleted, an error in the class MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be raised. Errors are raised using the default error handler (see Section 13.7).

### 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) |

**C binding**

```
int MPI_File_set_size(MPI_File fh, MPI_Offset size)
```

**F08 binding**

```
MPI_File_set_size(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_SET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

MPI_FILE_SET_SIZE resizes the file associated with the file handle fh. size is measured in bytes from the beginning of the file. MPI_FILE_SET_SIZE is collective; all processes in the group must pass identical values for size.

If size is smaller than the current file size, the file is truncated at the position defined by size. The implementation is free to deallocate file blocks located beyond this position.

If size is larger than the current file size, the file size becomes size. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (those locations with displacements between old file size and size) are undefined. It is implementation dependent whether the MPI_FILE_SET_SIZE routine allocates file space — use MPI_FILE_PREALLOCATE to force file space to be reserved.

MPI_FILE_SET_SIZE does not affect the individual file pointers or the shared file 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 valid, 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).

### 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) |

**C binding**

```
int MPI_File_preallocate(MPI_File fh, MPI_Offset size)
```

**F08 binding**

```
MPI_File_preallocate(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_PREALLOCATE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) 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) |

**C binding**
```
int MPI_File_get_size(MPI_File fh, MPI_Offset *size)
```

**F08 binding**
```
MPI_File_get_size(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**
```
MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE
```

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

### 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) |

**C binding**
```
int MPI_File_get_group(MPI_File fh, MPI_Group *group)
```

**F08 binding**
```
MPI_File_get_group(fh, group, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**
```
MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
    INTEGER FH, GROUP, IERROR
```

MPI_FILE_GET_GROUP returns a duplicate of the group of the communicator used to open the file associated with fh. The group is returned in group. The user is responsible for freeing group.

MPI_FILE_GET_AMODE(fh, amode)

| | | |
|---|---|---|
| IN | fh | file handle (handle) |
| OUT | amode | file access mode used to open the file (integer) |

**C binding**

```
int MPI_File_get_amode(MPI_File fh, int *amode)
```

**F08 binding**

```
MPI_File_get_amode(fh, amode, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER, INTENT(OUT) ::  amode
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR
```

MPI_FILE_GET_AMODE returns, in amode, the access mode of the file associated with fh.

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

```
      SUBROUTINE BIT_QUERY(TEST_BIT, MAX_BIT, AMODE, BIT_FOUND)
!
!   TEST IF THE INPUT TEST_BIT IS SET IN THE INPUT AMODE
!   IF SET, RETURN 1 IN BIT_FOUND, 0 OTHERWISE
!
      INTEGER TEST_BIT, AMODE, BIT_FOUND, CP_AMODE, HIFOUND
      BIT_FOUND = 0
      CP_AMODE = AMODE
 100  CONTINUE
      LBIT = 0
      HIFOUND = 0
      DO 20 L = MAX_BIT, 0, -1
         MATCHER = 2**L
         IF (CP_AMODE .GE. MATCHER .AND. HIFOUND .EQ. 0) THEN
            HIFOUND = 1
            LBIT = MATCHER
            CP_AMODE = CP_AMODE - MATCHER
         END IF
  20  CONTINUE
      IF (HIFOUND .EQ. 1 .AND. LBIT .EQ. TEST_BIT) BIT_FOUND = 1
      IF (BIT_FOUND .EQ. 0 .AND. HIFOUND .EQ. 1 .AND. &
          CP_AMODE .GT. 0) GO TO 100
      END
```

This routine could be called successively to decode amode, one bit at a time. For example, the following code fragment would check for MPI_MODE_RDONLY.

```
CALL BIT_QUERY(MPI_MODE_RDONLY, 30, AMODE, BIT_FOUND)
IF (BIT_FOUND .EQ. 1) THEN
   PRINT *, ' FOUND READ-ONLY BIT IN AMODE=', AMODE
ELSE
   PRINT *, ' READ-ONLY BIT NOT FOUND IN AMODE=', AMODE
END IF
```

### 13.2.8 File Info

Hints specified via info (see Chapter 9) allow a user to provide information such as file access patterns and file system specifics to direct optimization. Providing hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. An implementation is free to ignore all hints; however, applications must comply with any info hints they provide that are used by the MPI implementation (i.e., are returned by a call to MPI_FILE_GET_INFO) and that place a restriction on the behavior of the application. Hints are specified on a per file basis, in MPI_FILE_OPEN, MPI_FILE_DELETE, MPI_FILE_SET_VIEW, and MPI_FILE_SET_INFO, via the opaque info object. When an info object that specifies a subset of valid hints is passed to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO, there will be no effect on previously set or defaulted hints that the info does not specify.

> *Advice to implementors.* It may happen that a program is coded with hints for one system, and later executes on another system that does not support these hints. In general, unsupported hints should simply be ignored. 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) |

**C binding**

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

**F08 binding**

```
MPI_File_set_info(fh, info, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_SET_INFO(FH, INFO, IERROR)
    INTEGER FH, INFO, IERROR
```

MPI_FILE_SET_INFO updates the hints of the file associated with fh using the hints provided in info. This operation has no effect on previously set or defaulted hints that are not specified by info. It also has no effect on previously set or defaulted hints that are specified by info, but are ignored by the MPI implementation in this call to MPI_FILE_SET_INFO. MPI_FILE_SET_INFO is a collective routine. The info object may be different on each process, but any info entries that an implementation requires to be the same on all processes must appear with the same value in each process's info object.

*Advice to users.* 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. An implementation may also be unable to update certain info hints in a call to MPI_FILE_SET_VIEW or MPI_FILE_SET_INFO. MPI_FILE_GET_INFO can be used to determine whether info changes were ignored by the implementation. (*End of advice to users.*)

MPI_FILE_GET_INFO(fh, info_used)

| | | |
|---|---|---|
| IN | fh | file handle (handle) |
| OUT | info_used | new info object (handle) |

**C binding**

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)
```

**F08 binding**

```
MPI_File_get_info(fh, info_used, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Info), INTENT(OUT) :: info_used
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR
```

MPI_FILE_GET_INFO returns a new info object containing the hints of the file associated with fh. The current setting of all hints related to this file is returned in info_used. An MPI implementation is required to return all hints that are supported by the implementation and have default values specified; any user-supplied hints that were not ignored by the implementation; and any additional hints that were set by the implementation. If no such hints exist, a handle to a newly created info object is returned that contains no key/value pairs. The user is responsible for freeing info_used via MPI_INFO_FREE.

### 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 Chapter 9.)

These hints mainly affect access patterns and the layout of data on parallel I/O devices. For each hint name introduced, we describe the purpose of the hint, and the type of the hint value. The "[**SAME**]" annotation specifies that the hint values provided by all participating processes must be identical; otherwise the program is erroneous. In addition, some hints are context dependent, and are only used by an implementation at specific times (e.g., file_perm is only useful during file creation).

access_style **(comma separated list of strings):** This hint specifies the manner in which the file will be accessed until the file is closed or until the access_style key value is altered. The hint value is a comma separated list of the following: read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random.

collective_buffering **(boolean) [SAME]:** This hint specifies whether the application may benefit from collective buffering. Collective buffering is an optimization performed on collective accesses. Accesses to the file are performed on behalf of all processes in the group by a number of target nodes. These target nodes coalesce small requests into large disk accesses. Valid values for this key are true and false. Collective buffering parameters are further directed via additional hints: cb_block_size, cb_buffer_size, and cb_nodes.

cb_block_size **(integer) [SAME]:** This hint specifies the block size to be used for collective buffering file access. *Target nodes* access data in chunks of this size. The chunks are distributed among target nodes in a round-robin (cyclic) pattern.

cb_buffer_size **(integer) [SAME]:** This hint specifies the total buffer space that can be used for collective buffering on each target node, usually a multiple of cb_block_size.

cb_nodes **(integer) [SAME]:** This hint specifies the number of target nodes to be used for collective buffering.

chunked **(comma separated list of integers) [SAME]:** This hint specifies that the file consists of a multidimentional array that is often accessed by subarrays. The value for this hint is a comma separated list of array dimensions, starting from the most significant one (for an array stored in row-major order, as in C, the most significant dimension is the first one; for an array stored in column-major order, as in Fortran, the most significant dimension is the last one, and array dimensions should be reversed).

chunked_item **(comma separated list of integers) [SAME]:** This hint specifies the size of each array entry, in bytes.

chunked_size **(comma separated list of integers) [SAME]:** This hint specifies the dimensions of the subarrays. This is a comma separated list of array dimensions, starting from the most significant one.

filename **(string):** This hint specifies the file name used when the file was opened. If the implementation is capable of returning the file name of an open file, it will be returned using this key by MPI_FILE_GET_INFO. This key is ignored when passed to MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO, and MPI_FILE_DELETE.

file_perm **(string) [SAME]:** This hint specifies the file permissions to use for file creation. Setting this hint is only useful when passed to MPI_FILE_OPEN with an amode that includes MPI_MODE_CREATE. The set of valid values for this key is implementation dependent.

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

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

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

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

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

## 13.3 File Views

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

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

**C binding**

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

**F08 binding**

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  disp
    TYPE(MPI_Datatype), INTENT(IN) ::  etype, filetype
    CHARACTER(LEN=*), INTENT(IN) ::  datarep
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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
```

The MPI_FILE_SET_VIEW routine changes the process's view of the data in the file. The start of the view is set to disp; the type of data is set to etype; the distribution of data to processes is set to filetype; and the representation of data in the file is set to datarep. In addition, MPI_FILE_SET_VIEW resets the individual file pointers and the shared file pointer to zero. MPI_FILE_SET_VIEW is collective; the values for datarep and the extents of etype in the file data representation must be identical on all processes in the group; values for disp, filetype, and info may vary. The datatypes passed in etype and filetype must be committed.

The etype always specifies the data layout in the file. If etype is a portable datatype (see Section 2.4), the extent of etype is computed by scaling any displacements in the datatype to match the file data representation. If etype is not a portable datatype, no scaling is done when computing the extent of etype. The user must be careful when using nonportable etypes in heterogeneous environments; see Section 13.5.1 for further details.

If MPI_MODE_SEQUENTIAL mode was specified when the file was opened, the special displacement MPI_DISPLACEMENT_CURRENT must be passed in disp. This sets the displacement to the current position of the shared file pointer. MPI_DISPLACEMENT_CURRENT is invalid unless the amode for the file has MPI_MODE_SEQUENTIAL set.

*Rationale.* For some sequential files, such as those corresponding to magnetic tapes or streaming network connections, the *displacement* may not be meaningful. MPI_DISPLACEMENT_CURRENT allows the view to be changed for these types of files. (*End of rationale.*)

*Advice to implementors.* It is expected that a call to MPI_FILE_SET_VIEW will immediately follow MPI_FILE_OPEN in numerous instances. A high-quality implementation will ensure that this behavior is efficient. (*End of advice to implementors.*)

The disp displacement argument specifies the position (absolute offset in bytes from the beginning of the file) where the view begins.

*Advice to users.* disp can be used to skip headers or when the file includes a sequence of data segments that are to be accessed in different patterns (see Figure 13.3). Separate views, each using a different displacement and filetype, can be used to access each segment.

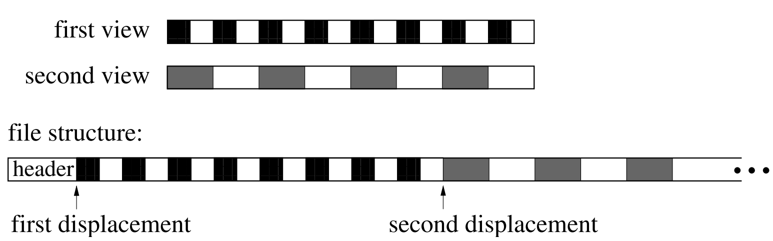


Figure 13.3: Displacements

(*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 non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes.

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

A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. These displacements are not required to be distinct, but they cannot be negative, and they must be monotonically nondecreasing.

If the file is opened for writing, neither the etype nor the filetype is permitted to contain overlapping regions. This restriction is equivalent to the "datatype used in a receive cannot specify overlapping regions" restriction for communication. Note that filetypes from different processes may still overlap each other.

If a filetype has holes in it, then the data in the holes is inaccessible to the calling process. However, the disp, etype, and filetype arguments can be changed via future calls to MPI_FILE_SET_VIEW to access a different part of the file.

It is erroneous to use absolute addresses in the construction of the etype and filetype.

The info argument is used to provide information regarding file access patterns and file system specifics to direct optimization (see Section 13.2.8). The constant MPI_INFO_NULL refers to the null info and can be used when no info needs to be specified.

The datarep argument is a string that specifies the representation of data in the file. See the file interoperability section (Section 13.5) for details and a discussion of valid values.

The user is responsible for ensuring that all nonblocking requests and split collective operations on fh have been completed before calling MPI_FILE_SET_VIEW — otherwise, the call to MPI_FILE_SET_VIEW is erroneous.

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

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

**C binding**

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

**F08 binding**

```
MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) ::  disp
    TYPE(MPI_Datatype), INTENT(OUT) ::  etype, filetype
```

```
    CHARACTER(LEN=*), INTENT(OUT) ::  datarep
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, IERROR)
    INTEGER FH, ETYPE, FILETYPE, IERROR
    CHARACTER*(*) DATAREP
    INTEGER(KIND=MPI_OFFSET_KIND) DISP
```

MPI_FILE_GET_VIEW returns the process's view of the data in the file. The current value of the displacement is returned in disp. The etype and filetype are new datatypes with typemaps equal to the typemaps of the current etype and filetype, respectively.

The data representation is returned in datarep. The user is responsible for ensuring that datarep is large enough to hold the returned data representation string. The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING.

In addition, if a portable datatype was used to set the current view, then the corresponding datatype returned by MPI_FILE_GET_VIEW is also a portable datatype. If etype or filetype are derived datatypes, the user is responsible for freeing them. The etype and filetype returned are both in a committed state.

## 13.4 Data Access

### 13.4.1 Data Access Routines

Data is moved between files and processes by issuing read and write calls. There are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (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.

| **positioning** | **synchronism** | **coordination** | |
|---|---|---|---|
| | | *noncollective* | *collective* |
| *explicit offsets* | *blocking* | MPI_FILE_READ_AT<br>MPI_FILE_WRITE_AT | MPI_FILE_READ_AT_ALL<br>MPI_FILE_WRITE_AT_ALL |
| | *nonblocking* | MPI_FILE_IREAD_AT<br>MPI_FILE_IWRITE_AT | MPI_FILE_IREAD_AT_ALL<br>MPI_FILE_IWRITE_AT_ALL |
| | *split collective* | N/A | MPI_FILE_READ_AT_ALL_BEGIN<br>MPI_FILE_READ_AT_ALL_END<br>MPI_FILE_WRITE_AT_ALL_BEGIN<br>MPI_FILE_WRITE_AT_ALL_END |
| *individual file pointers* | *blocking* | MPI_FILE_READ<br>MPI_FILE_WRITE | MPI_FILE_READ_ALL<br>MPI_FILE_WRITE_ALL |
| | *nonblocking* | MPI_FILE_IREAD<br>MPI_FILE_IWRITE | MPI_FILE_IREAD_ALL<br>MPI_FILE_IWRITE_ALL |
| | *split collective* | N/A | MPI_FILE_READ_ALL_BEGIN<br>MPI_FILE_READ_ALL_END<br>MPI_FILE_WRITE_ALL_BEGIN<br>MPI_FILE_WRITE_ALL_END |
| *shared file pointer* | *blocking* | MPI_FILE_READ_SHARED<br>MPI_FILE_WRITE_SHARED | MPI_FILE_READ_ORDERED<br>MPI_FILE_WRITE_ORDERED |
| | *nonblocking* | MPI_FILE_IREAD_SHARED<br>MPI_FILE_IWRITE_SHARED | N/A |
| | *split collective* | N/A | MPI_FILE_READ_ORDERED_BEGIN<br>MPI_FILE_READ_ORDERED_END<br>MPI_FILE_WRITE_ORDERED_BEGIN<br>MPI_FILE_WRITE_ORDERED_END |

Table 13.1: Data access routines

POSIX `read()`/`fread()` and `write()`/`fwrite()` are blocking, noncollective operations and use individual file pointers. The MPI equivalents are MPI_FILE_READ and

MPI_FILE_WRITE.

Implementations of data access routines may buffer data to improve performance. This does not affect reads, as the data is always available in the user's buffer after a read operation completes. For writes, however, the MPI_FILE_SYNC routine provides the only guarantee that data has been transferred to the storage device.

#### Positioning

MPI provides three types of positioning for data access routines: **explicit offsets**, **individual file pointers**, and **shared file pointers**. The different positioning methods may be mixed within the same program and do not affect each other.

The data access routines that accept explicit offsets contain _AT in their name (e.g., MPI_FILE_WRITE_AT). Explicit offset operations perform data access at the file position given directly as an argument — no file pointer is used nor updated. Note that this is not equivalent to an atomic seek-and-read or seek-and-write operation, as no "seek" is issued. Operations with explicit offsets are described in Section 13.4.2.

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. 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.

The main semantic issues with MPI-maintained file pointers are how and when they are updated by I/O operations. In general, each I/O operation leaves the file pointer pointing to the next data item after the last one that is accessed by the operation. In a nonblocking or split collective operation, the pointer is updated by the call that initiates the I/O, possibly before the access completes.

More formally,

$$new\_file\_offset = old\_file\_offset + \frac{elements(datatype)}{elements(etype)} \times count$$

where *count* is the number of *datatype* items to be accessed, $elements(X)$ is the number of predefined datatypes in the typemap of $X$, and *old_file_offset* is the value of the implicit offset before the call. The file position, *new_file_offset*, is in terms of a count of etypes relative to the current view.

#### 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 of and into the user's buffer to proceed concurrently with computation. A separate *request complete* call (MPI_WAIT, MPI_TEST, or any of their variants) is needed to complete the I/O request, i.e., to confirm that the data has been read or written and that it is safe for the user to reuse the buffer. The nonblocking versions of the routines are named MPI_FILE_IXXX, where the I stands for immediate.

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).

### Coordination

Every noncollective data access routine MPI_FILE_XXX has a collective counterpart. For most routines, this counterpart is MPI_FILE_XXX_ALL or a pair of MPI_FILE_XXX_BEGIN and MPI_FILE_XXX_END. The counterparts to the MPI_FILE_XXX_SHARED routines are MPI_FILE_XXX_ORDERED.

The completion of a noncollective call only depends on the activity of the calling process. However, the completion of a collective call (which must be called by all members of the process group) may depend on the activity of the other processes participating in the collective call. See Section 13.6.4 for rules on semantics of collective calls.

Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimization.

### Data Access Conventions

Data is moved between files and processes by calling read and write routines. Read routines move data from a file into memory. Write routines move data from memory into a file. The file is designated by a file handle, fh. The location of the file data is specified by an offset into the current view. The data in memory is specified by a triple: buf, count, and datatype. Upon completion, the amount of data accessed by the calling process is returned in a status.

An offset designates the starting position in the file for an access. The offset is always in etype units relative to the current view. Explicit offset routines pass offset as an argument (negative values are erroneous). The file pointer routines use implicit offsets maintained by MPI.

A data access routine attempts to transfer (read or write) count data items of type datatype between the user's buffer buf and the file. The datatype passed to the routine must be a committed datatype. The layout of data in memory corresponding to buf, count, datatype is interpreted the same way as in MPI communication functions; see Section 3.2.2 and Section 4.1.11. The data is accessed from those parts of the file specified by the current view (Section 13.3). The type signature of datatype must match the type signature of some number of contiguous copies of the etype of the current view. As in a receive, it is erroneous to specify a datatype for reading that contains overlapping regions (areas of memory which would be stored into more than once).

The nonblocking data access routines indicate that MPI can start a data access and associate a request handle, request, with the I/O operation. Nonblocking operations are completed via MPI_TEST, MPI_WAIT, or any of their variants.

Data access operations, when completed, return the amount of data accessed in status.

> *Advice to users.* To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in Sections 17.1.10–17.1.20. (*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 (or

MPI_GET_ELEMENTS_X), 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. The status can be passed to MPI_TEST_CANCELLED 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) |

**C binding**

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

**F08 binding**

```
MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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_AT reads a file beginning at the position specified by offset.

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

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

**C binding**

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

**F08 binding**

```
MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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_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) |

**C binding**

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

**F08 binding**

```
MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
```

```
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE_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_WRITE_AT writes a file beginning at the position specified by offset.

MPI_FILE_WRITE_AT_ALL(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) |

**C binding**

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

**F08 binding**

```
MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE_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_WRITE_AT_ALL is a collective version of the blocking MPI_FILE_WRITE_AT interface.

MPI_FILE_IREAD_AT(fh, offset, buf, count, datatype, request)

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

**C binding**

```
int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
             MPI_Datatype datatype, MPI_Request *request)
```

**F08 binding**

```
MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

MPI_FILE_IREAD_AT is a nonblocking version of the MPI_FILE_READ_AT interface.

MPI_FILE_IREAD_AT_ALL(fh, offset, buf, count, datatype, request)

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

**C binding**

```
int MPI_File_iread_at_all(MPI_File fh, MPI_Offset offset, void *buf,
             int count, MPI_Datatype datatype, MPI_Request *request)
```

**F08 binding**

```
MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
```

```
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

MPI_FILE_IREAD_AT_ALL is a nonblocking version of MPI_FILE_READ_AT_ALL. See Section 13.6.5 for semantics of nonblocking collective file operations.

MPI_FILE_IWRITE_AT(fh, offset, buf, count, datatype, request)

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

**C binding**

```
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
              int count, MPI_Datatype datatype, MPI_Request *request)
```

**F08 binding**

```
MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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_AT is a nonblocking version of the MPI_FILE_WRITE_AT interface.

MPI_FILE_IWRITE_AT_ALL(fh, offset, buf, count, datatype, request)

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

**C binding**

```
int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
             int count, MPI_Datatype datatype, MPI_Request *request)
```

**F08 binding**

```
MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_IWRITE_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

MPI_FILE_IWRITE_AT_ALL is a nonblocking version of MPI_FILE_WRITE_AT_ALL.

### 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, with the following modification:

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

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

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

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) |

**C binding**

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

**F08 binding**

```
MPI_File_read(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_FILE_READ reads a file using the individual file pointer.

**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.

      integer   bufsize, numread, totprocessed, status(MPI_STATUS_SIZE)
      parameter (bufsize=100)
      real      localbuffer(bufsize)
      integer (kind=MPI_OFFSET_KIND) zero

      zero = 0

      call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
                         MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
      call MPI_FILE_SET_VIEW(myfh, zero, 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) |

**C binding**

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

**F08 binding**

```
MPI_File_read_all(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_READ_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |

**C binding**

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

**F08 binding**

```
MPI_File_write(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_FILE_WRITE writes a file using the individual file pointer.

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) |

**C binding**

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

**F08 binding**

```
MPI_File_write_all(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_WRITE_ALL(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_FILE_WRITE_ALL is a collective version of the blocking MPI_FILE_WRITE interface.

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) |

**C binding**

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

**F08 binding**

```
MPI_File_iread(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

MPI_FILE_IREAD is a nonblocking version of the MPI_FILE_READ interface.

**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)
      integer (kind=MPI_OFFSET_KIND) zero

      zero = 0
      call MPI_FILE_OPEN(MPI_COMM_WORLD, 'myoldfile', &
                         MPI_MODE_RDONLY, MPI_INFO_NULL, myfh, ierr)
      call MPI_FILE_SET_VIEW(myfh, zero, 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_IREAD_ALL(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) |

**C binding**

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

**F08 binding**

```
MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

MPI_FILE_IREAD_ALL is a nonblocking version of MPI_FILE_READ_ALL.

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

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

**C binding**

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

**F08 binding**

```
MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

MPI_FILE_IWRITE is a nonblocking version of the MPI_FILE_WRITE interface.

MPI_FILE_IWRITE_ALL(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) |

**C binding**

```
int MPI_File_iwrite_all(MPI_File fh, const void *buf, int count,
             MPI_Datatype datatype, MPI_Request *request)
```

**F08 binding**

```
MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Request), INTENT(OUT) ::  request
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

MPI_FILE_IWRITE_ALL is a nonblocking version of MPI_FILE_WRITE_ALL.

MPI_FILE_SEEK(fh, offset, whence)

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

**C binding**

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

**F08 binding**

```
MPI_File_seek(fh, offset, whence, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    INTEGER, INTENT(IN) :: whence
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

MPI_FILE_SEEK updates the individual file pointer according to whence, which has the following possible values:

- MPI_SEEK_SET: the pointer is set to offset
- MPI_SEEK_CUR: the pointer is set to the current pointer position plus offset
- MPI_SEEK_END: the pointer is set to the end of file plus offset

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

MPI_FILE_GET_POSITION(fh, offset)

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

**C binding**

```
int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)
```

**F08 binding**

```
MPI_File_get_position(fh, offset, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

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

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) |

**C binding**

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

**F08 binding**

```
MPI_File_get_byte_offset(fh, offset, disp, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP
```

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, 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) |

**C binding**

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

**F08 binding**

```
MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_READ_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |

**C binding**

```
int MPI_File_write_shared(MPI_File fh, const void *buf, int count,
             MPI_Datatype datatype, MPI_Status *status)
```

**F08 binding**

```
MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE_SHARED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |

**C binding**

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

**F08 binding**

```
MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

MPI_FILE_IREAD_SHARED is a nonblocking version of the MPI_FILE_READ_SHARED interface.

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

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

**C binding**

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

**F08 binding**

```
MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

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

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) |

**C binding**

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

**F08 binding**

```
MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_READ_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |
| IN | datatype | datatype of each buffer element (handle) |
| OUT | status | status object (Status) |

**C binding**

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

**F08 binding**

```
MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE_ORDERED(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |

**C binding**

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

**F08 binding**

```
MPI_File_seek_shared(fh, offset, whence, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
```

```
    INTEGER, INTENT(IN) :: whence
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_SEEK_SHARED(FH, OFFSET, WHENCE, IERROR)
    INTEGER FH, WHENCE, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

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) |

**C binding**

```
int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)
```

**F08 binding**

```
MPI_File_get_position_shared(fh, offset, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

MPI_FILE_GET_POSITION_SHARED returns, in offset, the current position of the shared 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_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.
- Begin calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls.
- End calls are collective over the group of processes that participated in the collective open and follow the ordering rules for collective calls. Each end call matches the preceding begin call for the same collective operation. When an "end" call is made, exactly one unmatched "begin" call for the same operation must precede it.
- An implementation is free to implement any split collective data access routine using the corresponding blocking collective routine when either the begin call (e.g., MPI_FILE_READ_ALL_BEGIN) or the end call (e.g., MPI_FILE_READ_ALL_END) is issued. The begin and end calls are provided to allow the user and MPI implementation to optimize the collective operation.
- Split collective operations do not match the corresponding regular collective operation. For example, in a single collective read operation, an MPI_FILE_READ_ALL on one process does not match an MPI_FILE_READ_ALL_BEGIN/ MPI_FILE_READ_ALL_END pair on another process.
- Split collective routines must specify a buffer in both the begin and end routines. By specifying the buffer that receives data in the end routine, we can avoid the problems described in "A Problem with Code Movements and Register Optimization," Section 17.1.17, but not all of the problems, such as those described in Sections 17.1.12, 17.1.13, and 17.1.16.
- No collective I/O operations are permitted on a file handle concurrently with a split collective access on that file handle (i.e., between the begin and end of the access). That is

```
MPI_File_read_all_begin(fh, ...);
...
MPI_File_read_all(fh, ...);
...
MPI_File_read_all_end(fh, ...);
```

is erroneous.

- In a multithreaded implementation, any split collective begin and end operation called by a process must be called from the same thread. This restriction is made to simplify the implementation in the multithreaded case. (Note that we have already disallowed having two threads begin a split collective operation on the same file handle since only one split collective operation can be active on a file handle at any time.)

The arguments for these routines have the same meaning as for the equivalent collective versions (e.g., the argument definitions for MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END are equivalent to the arguments for MPI_FILE_READ_ALL). The begin routine (e.g., MPI_FILE_READ_ALL_BEGIN) begins a split collective operation that, when completed with the matching end routine (i.e., MPI_FILE_READ_ALL_END) produces the result as defined for the equivalent collective routine (i.e., MPI_FILE_READ_ALL).

For the purpose of consistency semantics (Section 13.6.1), a matched pair of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access.

MPI_FILE_READ_AT_ALL_BEGIN(fh, offset, buf, count, datatype)

| | | |
|---|---|---|
| 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) |

**C binding**

```
int MPI_File_read_at_all_begin(MPI_File fh, MPI_Offset offset, void *buf,
             int count, MPI_Datatype datatype)
```

**F08 binding**

```
MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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)

| | | |
|---|---|---|
| IN | fh | file handle (handle) |
| OUT | buf | initial address of buffer (choice) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_File_read_at_all_end(MPI_File fh, void *buf, MPI_Status *status)
```

**F08 binding**

```
MPI_File_read_at_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_FILE_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) |

**C binding**

```
int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
              const void *buf, int count, MPI_Datatype datatype)
```

**F08 binding**

```
MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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)

| | | |
|---|---|---|
| INOUT | fh | file handle (handle) |
| IN | buf | initial address of buffer (choice) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
             MPI_Status *status)
```

**F08 binding**

```
MPI_File_write_at_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE_AT_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_FILE_READ_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) |

**C binding**

```
int MPI_File_read_all_begin(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype)
```

**F08 binding**

```
MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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)

| | | |
|---|---|---|
| INOUT | fh | file handle (handle) |
| OUT | buf | initial address of buffer (choice) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_File_read_all_end(MPI_File fh, void *buf, MPI_Status *status)
```

**F08 binding**

```
MPI_File_read_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_READ_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |

**C binding**

```
int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
             MPI_Datatype datatype)
```

**F08 binding**

```
MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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)

| | | |
|---|---|---|
| INOUT | fh | file handle (handle) |
| IN | buf | initial address of buffer (choice) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_File_write_all_end(MPI_File fh, const void *buf,
             MPI_Status *status)
```

**F08 binding**

```
MPI_File_write_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE_ALL_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |

**C binding**

```
int MPI_File_read_ordered_begin(MPI_File fh, void *buf, int count,
             MPI_Datatype datatype)
```

**F08 binding**

```
MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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)

| | | |
|---|---|---|
| INOUT | fh | file handle (handle) |
| OUT | buf | initial address of buffer (choice) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_File_read_ordered_end(MPI_File fh, void *buf, MPI_Status *status)
```

**F08 binding**

```
MPI_File_read_ordered_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_READ_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

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) |

**C binding**

```
int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
             MPI_Datatype datatype)
```

**F08 binding**

```
MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
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)

| | | |
|---|---|---|
| INOUT | fh | file handle (handle) |
| IN | buf | initial address of buffer (choice) |
| OUT | status | status object (Status) |

**C binding**

```
int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
             MPI_Status *status)
```

**F08 binding**

```
MPI_File_write_ordered_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_WRITE_ORDERED_END(FH, BUF, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, STATUS(MPI_STATUS_SIZE), IERROR
```

## 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) as well as the data conversion functions (Section 13.5.3).

Interoperability within a single MPI environment (which could be considered "operability") ensures that file data written by one MPI process can be read by any other MPI process, subject to the consistency constraints (see Section 13.6.1), provided that it would have been possible to start the two processes simultaneously and have them reside in a single MPI_COMM_WORLD. Furthermore, both processes must see the same data values at every absolute byte offset in the file for which data was written.

This single environment file interoperability implies that file data is accessible regardless of the number of processes.

There are three aspects to file interoperability:

- transferring the bits,
- converting between different file structures, and
- converting between different machine representations.

The first two aspects of file interoperability are beyond the scope of this standard, as both are highly machine dependent. However, transferring the bits of a file into and out of the MPI environment (e.g., by writing a file to tape) is required to be supported by all MPI implementations. In particular, an implementation must specify how familiar 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). 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 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. The data conversion rules for communication also apply to these conversions (see Section 3.3.2). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation.

This data representation has several advantages. First, all processes reading the file in a heterogeneous MPI environment will automatically have the data converted to their respective native representations. Second, the file can be exported from one MPI environment and imported into any other MPI environment with the guarantee that the second environment will be able to read all the data in the file.

The disadvantage of this data representation is that data precision and I/O performance may be lost in data type conversions.

*Advice to implementors.* When implementing read and write operations on top of MPI message-passing, the message data should be converted to and from the "external32" representation in the client, and sent as type MPI_BYTE. This will avoid possible double data type conversions and the associated further loss of precision and performance. (*End of advice to implementors.*)

### 13.5.1 Datatypes for File Interoperability

If the file data representation is other than "native," care must be taken in constructing etypes and filetypes. Any of the datatype constructor functions may be used; however, for those functions that accept displacements in bytes, the displacements must be specified in terms of their values in the file for the file data representation being used. MPI will interpret these byte displacements as is; no scaling will be done. The function MPI_FILE_GET_TYPE_EXTENT can be used to calculate the extents of datatypes in the file. For etypes and filetypes that are portable datatypes (see Section 2.4), MPI will scale any displacements in the datatypes to match the file data representation. Datatypes passed as arguments to read/write routines specify the data layout in memory; therefore, they must always be constructed using displacements corresponding to displacements in memory.

*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_TYPE_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 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) 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) |

**C binding**

```
int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
             MPI_Aint *extent)
```

**F08 binding**

```
MPI_File_get_type_extent(fh, datatype, extent, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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), 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). (*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 [37] 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 C `_Bool`, Fortran `LOGICAL`, and C++ `bool`, `0` implies false and nonzero implies true. C `float _Complex`, `double _Complex`, and `long double _Complex`, Fortran `COMPLEX` and `DOUBLE COMPLEX`, and other complex types are represented by a pair of floating point format values for the real and imaginary components. Characters are in ISO 8859-1 format [38]. Wide characters (of type MPI_WCHAR) are in Unicode format [58].

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 [37], 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 https://www.ietf.org/rfc/rfc1832.txt). (*End of advice to implementors.*)

All data is byte aligned, regardless of type. All data items are stored contiguously in the file (if the file view is contiguous).

*Advice to implementors.* All bytes of `LOGICAL` and `bool` must be checked to determine the value. (*End of advice to implementors.*)

*Advice to users.* The type MPI_PACKED is treated as bytes and is not converted. The user should be aware that MPI_PACK has the option of placing a header in the beginning of the pack buffer. (*End of advice to users.*)

The sizes 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 17.1.9, page 643.

*Advice to implementors.* When converting a larger size integer to a smaller size integer, only the least 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.

| Type | Length |
|---|---|
| MPI_PACKED | 1 |
| MPI_BYTE | 1 |
| MPI_CHAR | 1 |
| MPI_UNSIGNED_CHAR | 1 |
| MPI_SIGNED_CHAR | 1 |
| MPI_WCHAR | 2 |
| MPI_SHORT | 2 |
| MPI_UNSIGNED_SHORT | 2 |
| MPI_INT | 4 |
| MPI_UNSIGNED | 4 |
| MPI_LONG | 4 |
| MPI_UNSIGNED_LONG | 4 |
| MPI_LONG_LONG_INT | 8 |
| MPI_UNSIGNED_LONG_LONG | 8 |
| MPI_FLOAT | 4 |
| MPI_DOUBLE | 8 |
| MPI_LONG_DOUBLE | 16 |
| | |
| MPI_C_BOOL | 1 |
| MPI_INT8_T | 1 |
| MPI_INT16_T | 2 |
| MPI_INT32_T | 4 |
| MPI_INT64_T | 8 |
| MPI_UINT8_T | 1 |
| MPI_UINT16_T | 2 |
| MPI_UINT32_T | 4 |
| MPI_UINT64_T | 8 |
| MPI_AINT | 8 |
| MPI_COUNT | 8 |
| MPI_OFFSET | 8 |
| MPI_C_COMPLEX | 2*4 |
| MPI_C_FLOAT_COMPLEX | 2*4 |
| MPI_C_DOUBLE_COMPLEX | 2*8 |
| MPI_C_LONG_DOUBLE_COMPLEX | 2*16 |
| | |
| MPI_CHARACTER | 1 |
| MPI_LOGICAL | 4 |
| MPI_INTEGER | 4 |
| MPI_REAL | 4 |
| MPI_DOUBLE_PRECISION | 8 |
| MPI_COMPLEX | 2*4 |
| MPI_DOUBLE_COMPLEX | 2*8 |

| Optional Type | Length |
|---|---|
| MPI_INTEGER1 | 1 |
| MPI_INTEGER2 | 2 |
| MPI_INTEGER4 | 4 |
| MPI_INTEGER8 | 8 |
| MPI_INTEGER16 | 16 |
| | |
| MPI_REAL2 | 2 |
| MPI_REAL4 | 4 |
| MPI_REAL8 | 8 |
| MPI_REAL16 | 16 |
| | |
| MPI_COMPLEX4 | 2*2 |
| MPI_COMPLEX8 | 2*4 |
| MPI_COMPLEX16 | 2*8 |
| MPI_COMPLEX32 | 2*16 |

| C++ Types | Length |
|---|---|
| MPI_CXX_BOOL | 1 |
| MPI_CXX_FLOAT_COMPLEX | 2*4 |
| MPI_CXX_DOUBLE_COMPLEX | 2*8 |
| MPI_CXX_LONG_DOUBLE_COMPLEX | 2*16 |

Table 13.2: “external32” sizes of predefined datatypes

### 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 |

**C binding**

```
int MPI_Register_datarep(const char *datarep,
             MPI_Datarep_conversion_function *read_conversion_fn,
             MPI_Datarep_conversion_function *write_conversion_fn,
             MPI_Datarep_extent_function *dtype_file_extent_fn,
             void *extra_state)
```

**F08 binding**

```
MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
             dtype_file_extent_fn, extra_state, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  datarep
    PROCEDURE(MPI_Datarep_conversion_function) ::  read_conversion_fn
    PROCEDURE(MPI_Datarep_conversion_function) ::  write_conversion_fn
    PROCEDURE(MPI_Datarep_extent_function) ::  dtype_file_extent_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_REGISTER_DATAREP(DATAREP, READ_CONVERSION_FN, WRITE_CONVERSION_FN,
             DTYPE_FILE_EXTENT_FN, EXTRA_STATE, IERROR)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER IERROR
```

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). The length of a data representation string is limited to the value of MPI_MAX_DATAREP_STRING. MPI_MAX_DATAREP_STRING must have a value of at least 64. No routines are provided to delete data representations and free the associated resources; it is not expected that an application will generate them in significant numbers.

### Extent Callback

```
typedef int MPI_Datarep_extent_function(MPI_Datatype datatype,
             MPI_Aint *file_extent, void *extra_state);
```

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
  ierror)
      TYPE(MPI_Datatype) ::  datatype
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extent, extra_state
      INTEGER ::  ierror
```

```
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE
```

The function dtype_file_extent_fn must return, in file_extent, the number of bytes required to store datatype in the file representation. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. MPI will only call this routine with predefined datatypes employed by the user.

### Datarep Conversion Functions

```
typedef int MPI_Datarep_conversion_function(void *userbuf,
             MPI_Datatype datatype, int count, void *filebuf,
             MPI_Offset position, void *extra_state);
```

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
  filebuf, position, extra_state, ierror)
      USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE ::  userbuf, filebuf
      TYPE(MPI_Datatype) ::  datatype
      INTEGER ::  count, ierror
      INTEGER(KIND=MPI_OFFSET_KIND) ::  position
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state
```

```
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 function read_conversion_fn must convert from file data representation to native representation. Before calling this routine, MPI allocates and fills filebuf with count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call. The function must copy all count data items from filebuf to userbuf in the distribution described by datatype, converting each data item from file representation to native representation. datatype will be equivalent to the datatype that the user passed to the read function. If the size of datatype is less than the size of the count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf. The conversion function must begin storing converted data at the location in userbuf specified by position into the (tiled) datatype.

*Advice to users.* Although the conversion functions have similarities to MPI_PACK and MPI_UNPACK, one should note the differences in the use of the arguments count and position. In the conversion functions, count is a count of data items (i.e., count of typemap entries of datatype), and position is an index into this typemap. In MPI_PACK, incount refers to the number of whole datatypes, and position is a number of bytes. (*End of advice to users.*)

*Advice to implementors.* A converted read operation could be implemented as follows:

1. Get file extent of all data items
2. Allocate a filebuf large enough to hold all count data items
3. Read data from file into filebuf
4. Call read_conversion_fn to convert data and place it into userbuf
5. Deallocate filebuf

(*End of advice to implementors.*)

If MPI cannot allocate a buffer large enough to hold all the data to be converted from a read operation, it may call the conversion function repeatedly using the same datatype and userbuf, and reading successive chunks of data to be converted in filebuf. For the first call (and in the case when all the data to be converted fits into filebuf), MPI will call the function with position set to zero. Data converted during this call will be stored in the userbuf according to the first count data items in datatype. Then in subsequent calls to the conversion function, MPI will increment the value in position by the count of items converted in the previous call, and the userbuf pointer will be unchanged.

*Rationale.* Passing the conversion function a position and one datatype for the transfer allows the conversion function to decode the datatype only once and cache an internal representation of it on the datatype. Then on subsequent calls, the conversion function can use the position to quickly find its place in the datatype and continue storing converted data where it left off at the end of the previous call. (*End of rationale.*)

*Advice to users.* Although the conversion function may usefully cache an internal representation on the datatype, it should not cache any state information specific to an ongoing conversion operation, since it is possible for the same datatype to be used concurrently in multiple conversion operations. (*End of advice to users.*)

The function write_conversion_fn must convert from native representation to file data representation. Before calling this routine, MPI allocates filebuf of a size large enough to hold count contiguous data items. The type of each data item matches the corresponding entry for the predefined datatype in the type signature of datatype. The function must copy count data items from userbuf in the distribution described by datatype, to a contiguous distribution in filebuf, converting each data item from native representation to file representation. If the size of datatype is less than the size of count data items, the conversion function must treat datatype as being contiguously tiled over the userbuf.

The function must begin copying at the location in userbuf specified by position into the (tiled) datatype. datatype will be equivalent to the datatype that the user passed to the write function. The function is passed, in extra_state, the argument that was passed to the MPI_REGISTER_DATAREP call.

The predefined constant MPI_CONVERSION_FN_NULL may be used as either write_conversion_fn or read_conversion_fn. In that case, MPI will not attempt to invoke write_conversion_fn or read_conversion_fn, respectively, but will perform the requested data access using the native data representation.

An MPI implementation must ensure that all data accessed is converted, either by using a filebuf large enough to hold all the requested data items or 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, or MPI_FILE_GET_TYPE_EXTENT is called by the user. dtype_file_extent_fn will only be passed predefined datatypes employed by the user. The conversion functions will only be passed datatypes equivalent to those that the user has passed to one of the routines noted above.

The conversion functions must be reentrant. User defined data representations are restricted to use byte alignment for all types. Furthermore, it is erroneous for the conversion functions to call any collective routines or to free datatype.

The conversion functions should return an error code. If the returned error code has a value other than MPI_SUCCESS, the implementation will raise an error in the class MPI_ERR_CONVERSION.

### 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, 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 17.1.9).
- For any given data item, the programs participating in the data accesses use compatible predefined types to write and read the data item.

User-defined data representations may be used to provide an implementation compatibility with another implementation's "native" or "internal" representation.

> *Advice to users.* Section 17.1.9 defines routines that support the use of matching datatypes in heterogeneous environments and contains examples illustrating their use. (*End of advice to users.*)

## 13.6 Consistency and Semantics

### 13.6.1 File Consistency

Consistency semantics define the outcome of multiple accesses to a single file. All file accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. Sequential consistency means the behavior of a set of operations will be as if the operations were performed in some serial order consistent with program order; each access appears atomic, although the exact ordering of accesses is unspecified. User-imposed consistency may be obtained using program order and calls to MPI_FILE_SYNC.

Let $FH_1$ be the set of file handles created from one particular collective open of the file $FOO$, and $FH_2$ be the set of file handles created from a different collective open of $FOO$. Note that nothing restrictive is said about $FH_1$ and $FH_2$: the sizes of $FH_1$ and $FH_2$ may be different, the groups of processes used for each open may or may not intersect, the file handles in $FH_1$ may be destroyed before those in $FH_2$ are created, etc. Consider the following three cases: a single file handle (e.g., $fh_1 \in FH_1$), two file handles created from a single collective open (e.g., $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$), and two file handles from different collective opens (e.g., $fh_1 \in FH_1$ and $fh_2 \in FH_2$).

For the purpose of consistency semantics, a matched pair (Section 13.4.5) of split collective data access operations (e.g., MPI_FILE_READ_ALL_BEGIN and MPI_FILE_READ_ALL_END) compose a single data access operation. Similarly, a nonblocking data access routine (e.g., MPI_FILE_IREAD) and the routine which completes the request (e.g., MPI_WAIT) also compose a single data access operation. For all cases below, these data access operations are subject to the same constraints as blocking data access operations.

*Advice to users.* For an MPI_FILE_IREAD and MPI_WAIT pair, the operation begins when MPI_FILE_IREAD is called and ends when MPI_WAIT returns. (*End of advice to users.*)

Assume that $A_1$ and $A_2$ are two data access operations. Let $D_1$ ($D_2$) be the set of absolute byte displacements of every byte accessed in $A_1$ ($A_2$). The two data accesses *overlap* if $D_1 \cap D_2 \neq \emptyset$. The two data accesses *conflict* if they overlap and at least one is a write access.

Let $SEQ_{fh}$ be a sequence of file operations on a single file handle, bracketed by MPI_FILE_SYNCs on that file handle. (Both opening and closing a file implicitly perform an MPI_FILE_SYNC.) $SEQ_{fh}$ is a "write sequence" if any of the data access operations in the sequence are writes or if any of the file manipulation operations in the sequence change the state of the file (e.g., MPI_FILE_SET_SIZE or MPI_FILE_PREALLOCATE). Given two sequences, $SEQ_1$ and $SEQ_2$, we say they are not *concurrent* if one sequence is guaranteed to completely precede the other (temporally).

The requirements for guaranteeing sequential consistency among all accesses to a particular file are divided into the three cases given below. If any of these requirements are not met, then the value of all data in that file is implementation dependent.

Case 1: $fh_1 \in FH_1$ All operations on $fh_1$ are sequentially consistent if atomic mode is set. If nonatomic mode is set, then all operations on $fh_1$ are sequentially consistent if they are either nonconcurrent, nonconflicting, or both.

Case 2: $fh_{1a} \in FH_1$ and $fh_{1b} \in FH_1$ Assume $A_1$ is a data access operation using $fh_{1a}$, and $A_2$ is a data access operation using $fh_{1b}$. If for any access $A_1$, there is no access $A_2$ that conflicts with $A_1$, then MPI guarantees sequential consistency.

However, unlike POSIX semantics, the default MPI semantics for conflicting accesses do not guarantee sequential consistency. If $A_1$ and $A_2$ conflict, sequential consistency can be guaranteed by either enabling atomic mode via the MPI_FILE_SET_ATOMICITY routine, or meeting the condition described in Case 3 below.

Case 3: $fh_1 \in FH_1$ and $fh_2 \in FH_2$ Consider access to a single file using file handles from distinct collective opens. In order to guarantee sequential consistency, MPI_FILE_SYNC must be used (both opening and closing a file implicitly perform an MPI_FILE_SYNC).

Sequential consistency is guaranteed among accesses to a single file if for any write sequence $SEQ_1$ to the file, there is no sequence $SEQ_2$ to the file which is *concurrent* with $SEQ_1$. To guarantee sequential consistency when there are write sequences, MPI_FILE_SYNC must be used together with a mechanism that guarantees nonconcurrency of the sequences.

See the examples in Section 13.6.11 for further clarification of some of these consistency semantics.

MPI_FILE_SET_ATOMICITY(fh, flag)

| | | |
|---|---|---|
| INOUT | fh | file handle (handle) |
| IN | flag | true to set atomic mode, false to set nonatomic mode (logical) |

**C binding**

```
int MPI_File_set_atomicity(MPI_File fh, int flag)
```

**F08 binding**

```
MPI_File_set_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    LOGICAL, INTENT(IN) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_FILE_SET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
```

Let $FH$ be the set of file handles created by one collective open. The consistency semantics for data access operations using $FH$ is set by collectively calling MPI_FILE_SET_ATOMICITY on $FH$. 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) |

**C binding**

```
int MPI_File_get_atomicity(MPI_File fh, int *flag)
```

**F08 binding**

```
MPI_File_get_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    LOGICAL, INTENT(OUT) ::  flag
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
```

MPI_FILE_GET_ATOMICITY returns the current consistency semantics for data access operations on the set of file handles created by one collective open. If flag is true, atomic mode is enabled; if flag is false, nonatomic mode is enabled.

MPI_FILE_SYNC(fh)

| | | |
|---|---|---|
| INOUT | fh | file handle (handle) |

**C binding**

```
int MPI_File_sync(MPI_File fh)
```

**F08 binding**

```
MPI_File_sync(fh, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_FILE_SYNC(FH, IERROR)
    INTEGER FH, IERROR
```

Calling MPI_FILE_SYNC with fh causes all previous writes to fh by the calling process to be transferred to the storage device. If other processes have made updates to the storage device, then all such updates become visible to subsequent reads of fh by the calling process. MPI_FILE_SYNC may be necessary to ensure sequential consistency in certain cases (see above).

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 implementors. In cases where the progress rules restrict possible implementation choices more than the interface specification alone, the progress rules take precedence.

All blocking routines must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error.

Nonblocking data access routines inherit the following progress rule from nonblocking point to point communication: a nonblocking write is equivalent to a nonblocking send for which a receive is eventually posted, and a nonblocking read is equivalent to a nonblocking receive for which a send is eventually posted.

Finally, an implementation is free to delay progress of collective routines until all processes in the group associated with the collective call have invoked the routine. Once all processes in the group have invoked the routine, the progress rule of the equivalent noncollective routine must be followed.

### 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 5.14.

Collective file operations are collective over a duplicate of the communicator used to open the file — this duplicate communicator is implicitly specified via the file handle argument. Different processes can pass different values for other arguments of a collective routine unless specified otherwise.

### 13.6.5 Nonblocking Collective File Operations

Nonblocking collective file operations are defined only for data access routines with explicit offsets and individual file pointers but not with shared file pointers.

Nonblocking collective file operations are subject to the same restrictions as blocking collective I/O operations. All processes belonging to the group of the communicator that was used to open the file must call collective I/O operations (blocking and nonblocking) in the same order. This is consistent with the ordering rules for collective operations in threaded environments. For a complete discussion, please refer to the semantics set forth in Section 5.14.

Nonblocking collective I/O operations do not match with blocking collective I/O operations. Multiple nonblocking collective I/O operations can be outstanding on a single file handle. High quality MPI implementations should be able to support a large number of pending nonblocking I/O operations.

All nonblocking collective I/O calls are local and return immediately, irrespective of the status of other processes. The call initiates the operation which may progress independently of any communication, computation, or I/O. The call returns a request handle, which must be passed to a completion call. Input buffers should not be modified and output buffers should not be accessed before the completion call returns. The same progress rules described for nonblocking collective operations apply for nonblocking collective I/O operations. For a complete discussion, please refer to the semantics set forth in Section 5.12.

### 13.6.6 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.*)

### 13.6.7 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.

### 13.6.8 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 with kind parameter MPI_OFFSET_KIND, which is defined in the `mpi_f08` module, the `mpi` module and the `mpif.h` include file.

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 17.2).

### 13.6.9 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).

### 13.6.10 File Size

The size of a file may be increased by writing to the file after the current end of file. The size may also be changed by calling MPI *size changing* routines, such as MPI_FILE_SET_SIZE. A call to a size changing routine does not necessarily change the file size. For example, calling MPI_FILE_PREALLOCATE with a size less than the current size does not change the size.

Consider a set of bytes that has been written to a file since the most recent call to a size changing routine, or since MPI_FILE_OPEN if no such routine has been called. Let the *high byte* be the byte in that set with the largest displacement. The file size is the larger of

- One plus the displacement of the high byte.
- The size immediately after the size changing routine, or MPI_FILE_OPEN, returned.

When applying consistency semantics, calls to MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE are considered writes to the file (which conflict with operations that access bytes at displacements between the old and new file sizes), and MPI_FILE_GET_SIZE is considered a read of the file (which overlaps with all accesses to the file).

*Advice to users.* Any sequence of operations containing the collective routines MPI_FILE_SET_SIZE and MPI_FILE_PREALLOCATE is a write sequence. As such, sequential consistency in nonatomic mode is not guaranteed unless the conditions in Section 13.6.1 are satisfied. (*End of advice to users.*)

File pointer update semantics (i.e., file pointers are updated by the amount accessed) are only guaranteed if file size changes are sequentially consistent.

*Advice to users.* Consider the following example. Given two operations made by separate processes to a file containing 100 bytes: an MPI_FILE_READ of 10 bytes and an MPI_FILE_SET_SIZE to 0 bytes. If the user does not enforce sequential consistency between these two operations, the file pointer may be updated by the amount requested (10 bytes) even if the amount accessed is zero bytes. (*End of advice to users.*)

### 13.6.11 Examples

The examples in this section illustrate the application of the MPI consistency and semantics guarantees. These address

- conflicting accesses on file handles obtained from a single collective open, and
- all accesses on file handles obtained from two separate collective opens.

The simplest way to achieve consistency for conflicting accesses is to obtain sequential consistency by setting atomic mode. For the code below, process 1 will read either 0 or 10 integers. If the latter, every element of `b` will be `5`. If nonatomic mode is set, the results of the read are undefined.

```
/* 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  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_File_sync(fh1);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_sync(fh1);
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);
```

The “sync-barrier-sync” construct is required because:

- The barrier ensures that the write on process 0 occurs before the read on process 1.
- The first sync guarantees that the data written by all processes is transferred to the storage device.
- The second sync guarantees that all data which has been transferred to the storage device is visible to all processes. (This does not affect process 0 in this example.)

The following program represents an erroneous attempt to achieve consistency by eliminating the apparently superfluous second “sync” call for each process.

```
/* ----------------  THIS EXAMPLE IS ERRONEOUS --------------- */
/* Process 0 */
int  i, a[10];
for (i=0;i<10;i++)
   a[i] = 5;

MPI_File_open(MPI_COMM_WORLD, "workfile",
              MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh0);
MPI_File_set_view(fh0, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write_at(fh0, 0, a, 10, MPI_INT, &status);
MPI_File_sync(fh0);
MPI_Barrier(MPI_COMM_WORLD);

/* Process 1 */
int  b[10];
MPI_File_open(MPI_COMM_WORLD, "workfile",
              MPI_MODE_RDWR | MPI_MODE_CREATE, MPI_INFO_NULL, &fh1);
MPI_File_set_view(fh1, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_Barrier(MPI_COMM_WORLD);
MPI_File_sync(fh1);
MPI_File_read_at(fh1, 0, b, 10, MPI_INT, &status);

/* ----------------  THIS EXAMPLE IS ERRONEOUS --------------- */
```

The above program also violates the MPI rule against out-of-order collective operations and will deadlock for implementations in which MPI_FILE_SYNC blocks.

> *Advice to users.* Some implementations may choose to implement MPI_FILE_SYNC as a 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:

```
int a = 4, b, TRUE=1;
MPI_File_open(MPI_COMM_WORLD, "myfile",
              MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
/* MPI_File_set_atomicity(fh, TRUE);   Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
MPI_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 sequential consistency, and the program will read either 2 or 4 into b. If atomic mode is not set, then sequential consistency is not guaranteed and the program may read something other than 2 or 4 due to the conflicting data access.

Similarly, the following code fragment does not order file accesses:

```
int a = 4, b;
MPI_File_open(MPI_COMM_WORLD, "myfile",
              MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
/* MPI_File_set_atomicity(fh, TRUE);   Use this to set atomic mode. */
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]);
MPI_Wait(&reqs[0], &status);
MPI_Wait(&reqs[1], &status);
```

If atomic mode is set, either 2 or 4 will be read into b. Again, MPI does not guarantee sequential consistency in nonatomic mode.

On the other hand, the following code fragment:

```
int a = 4, b;
MPI_File_open(MPI_COMM_WORLD, "myfile",
              MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_iwrite_at(fh, 10, &a, 1, MPI_INT, &reqs[0]);
MPI_Wait(&reqs[0], &status);
MPI_File_iread_at(fh,  10, &b, 1, MPI_INT, &reqs[1]);
MPI_Wait(&reqs[1], &status);
```

defines the same ordering as:

```
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_iwrite_all(fh,...);
MPI_File_iread_all(fh,...);
MPI_Waitall(...);
```

In addition, as mentioned in Section 13.6.5, nonblocking collective I/O operations have to be called in the same order on the file handle by all processes.

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).

## 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.

When MPI calls a user-defined error handler resulting from an error on a particular file handle, the first two arguments passed to the file error handler are the file handle and the error code. For I/O errors that are not associated with a valid file handle (e.g., in MPI_FILE_OPEN or MPI_FILE_DELETE), the first argument passed to the error handler is MPI_FILE_NULL.

I/O error handling differs from communication error handling in another important aspect. By default, the predefined error handler for file handles is MPI_ERRORS_RETURN. The default file error handler has two purposes: when a new file handle is created (by MPI_FILE_OPEN), the error handler for the new file handle is initially set to the default error handler, and I/O routines that have no valid file handle on which to raise an error (e.g., MPI_FILE_OPEN or MPI_FILE_DELETE) use the default file error handler. The default file error handler can be changed by specifying MPI_FILE_NULL as the fh argument to MPI_FILE_SET_ERRHANDLER. The current value of the default file error handler can be determined by passing MPI_FILE_NULL as the fh argument to MPI_FILE_GET_ERRHANDLER.

*Rationale.* For communication, the default error handler is inherited from MPI_COMM_WORLD. In I/O, there is no analogous "root" file handle from which default properties can be inherited. Rather than invent a new global file handle, the default file error handler is manipulated as if it were attached to MPI_FILE_NULL. (*End of rationale.*)

## 13.8 I/O Error Classes

The implementation dependent error codes returned by the I/O routines can be converted into the error classes defined in Table 13.3.

In addition, calls to routines in this chapter may raise errors in other MPI classes, such as MPI_ERR_TYPE.

## 13.9 Examples

### 13.9.1 Double Buffering with Split Collective I/O

This example shows how to overlap computation and output. The computation is performed by the function `compute_buffer()`.

```
/*=========================================================================
 *
 * Function:            double_buffer
 *
 * Synopsis:
 *     void double_buffer(
 *             MPI_File fh,                               ** IN
 *             MPI_Datatype buftype,                      ** IN
 *             int bufcount                               ** IN
 *     )
```

| | |
|---|---|
| MPI_ERR_FILE | Invalid file handle |
| MPI_ERR_NOT_SAME | Collective argument not identical on all processes, or collective routines called in a different order by different processes |
| MPI_ERR_AMODE | Error related to the amode passed to MPI_FILE_OPEN |
| MPI_ERR_UNSUPPORTED_DATAREP | Unsupported datarep passed to MPI_FILE_SET_VIEW |
| MPI_ERR_UNSUPPORTED_OPERATION | Unsupported operation, such as seeking on a file which supports sequential access only |
| MPI_ERR_NO_SUCH_FILE | File does not exist |
| MPI_ERR_FILE_EXISTS | File exists |
| MPI_ERR_BAD_FILE | Invalid file name (e.g., path name too long) |
| MPI_ERR_ACCESS | Permission denied |
| MPI_ERR_NO_SPACE | Not enough space |
| MPI_ERR_QUOTA | Quota exceeded |
| MPI_ERR_READ_ONLY | Read-only file or file system |
| MPI_ERR_FILE_IN_USE | File operation could not be completed, as the file is currently open by some process |
| MPI_ERR_DUP_DATAREP | Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP |
| MPI_ERR_CONVERSION | An error occurred in a user supplied data conversion function. |
| MPI_ERR_IO | Other I/O error |

Table 13.3: I/O Error Classes

```
 *
 * Description:
 *      Performs the steps to overlap computation with a collective write
 *      by using a double-buffering technique.
 *
 * 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
 *------------------------------------------------------------------------*/

/* 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);
    }

    /* 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

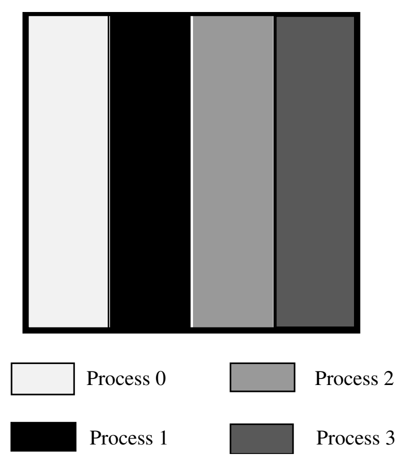


Figure 13.4: Example array file layout

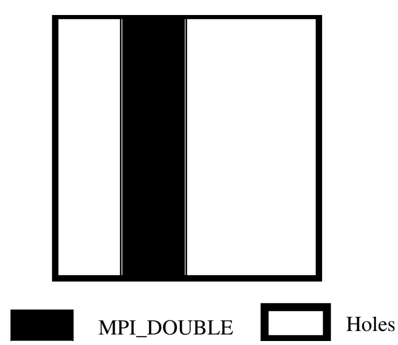


Figure 13.5: Example local array filetype for process 1

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):

```
double subarray[100][25];
MPI_Datatype filetype;
int sizes[2], subsizes[2], starts[2];
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sizes[0]=100; sizes[1]=100;
subsizes[0]=100; subsizes[1]=25;
starts[0]=0; starts[1]=rank*subsizes[1];

MPI_Type_create_subarray(2, sizes, subsizes, starts, MPI_ORDER_C,
                         MPI_DOUBLE, &filetype);
```

Or, equivalently in Fortran:

```
double precision subarray(100,25)
integer filetype, rank, ierror
integer sizes(2), subsizes(2), starts(2)

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
sizes(1)=100
sizes(2)=100
subsizes(1)=100
subsizes(2)=25
starts(1)=0
starts(2)=rank*subsizes(2)

call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, &
           MPI_ORDER_FORTRAN, MPI_DOUBLE_PRECISION,       &
           filetype, ierror)
```

The generated filetype will then describe the portion of the file contained within the process's subarray with holes for the space taken by the other processes. Figure 13.5 shows the filetype created for process 1.

# Chapter 14

# Deprecated Interfaces

## 14.1 Deprecated since MPI-2.0

The following function is deprecated and is superseded by MPI_COMM_CREATE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as that of the new function, except for the function name and a different behavior in the C/Fortran language interoperability, see Section 17.2.7. The language bindings are modified.

MPI_KEYVAL_CREATE(copy_fn, delete_fn, keyval, extra_state)

| | | |
|---|---|---|
| IN | copy_fn | Copy callback function for keyval (function) |
| IN | delete_fn | Delete callback function for keyval (function) |
| OUT | keyval | key value for future access (integer) |
| IN | extra_state | Extra state for callback functions |

**C binding**

```
int MPI_Keyval_create(MPI_Copy_function *copy_fn,
             MPI_Delete_function *delete_fn, int *keyval,
             void *extra_state)
```

**F binding**

```
MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
```

The copy_fn function is invoked when a communicator is duplicated by MPI_COMM_DUP. copy_fn should be of type MPI_Copy_function, which is defined as follows:

```
typedef int MPI_Copy_function(MPI_Comm oldcomm, int keyval,
                              void *extra_state, void *attribute_val_in,
                              void *attribute_val_out, int *flag)
```

A Fortran declaration for such a function is as follows:

```
For this routine, an interface within the mpi_f08 module was never defined.
```

```
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.

Analogous to copy_fn is a callback deletion function, defined as follows. The delete_fn function is invoked when a communicator is deleted by MPI_COMM_FREE or when a call is made explicitly to MPI_ATTR_DELETE. delete_fn should be of type MPI_Delete_function, which is defined as follows:

```
typedef int MPI_Delete_function(MPI_Comm comm, int keyval,
                                void *attribute_val, void *extra_state);
```

A Fortran declaration for such a function is as follows:

For this routine, an interface within the mpi_f08 module was never defined.

```
SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

delete_fn may be specified as MPI_NULL_DELETE_FN from either C or FORTRAN; MPI_NULL_DELETE_FN is a function that does nothing, other than returning MPI_SUCCESS. Note that MPI_NULL_DELETE_FN is also deprecated.

The following function is deprecated and is superseded by MPI_COMM_FREE_KEYVAL in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_KEYVAL_FREE(keyval)

| | | |
|---|---|---|
| INOUT | keyval | Frees the integer key value (integer) |

**C binding**

```
int MPI_Keyval_free(int *keyval)
```

**F binding**

```
MPI_KEYVAL_FREE(KEYVAL, IERROR)
    INTEGER KEYVAL, IERROR
```

The following function is deprecated and is superseded by MPI_COMM_SET_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_PUT(comm, keyval, attribute_val)

| | | |
|---|---|---|
| INOUT | comm | communicator to which attribute will be attached (handle) |
| IN | keyval | key value, as returned by MPI_KEYVAL_CREATE (integer) |
| IN | attribute_val | attribute value |

**C binding**

```
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
```

**F binding**

```
MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
```

The following function is deprecated and is superseded by MPI_COMM_GET_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_GET(comm, keyval, attribute_val, flag)

| | | |
|---|---|---|
| IN | comm | communicator to which attribute is attached (handle) |
| IN | keyval | key value (integer) |
| OUT | attribute_val | attribute value, unless flag = false |
| OUT | flag | true if an attribute value was extracted; false if no attribute is associated with the key (logical) |

**C binding**

```
int MPI_Attr_get(MPI_Comm comm, int keyval, void* attribute_val, int *flag)
```

**F binding**

```
MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG
```

The following function is deprecated and is superseded by MPI_COMM_DELETE_ATTR in MPI-2.0. The language independent definition of the deprecated function is the same as of the new function, except of the function name. The language bindings are modified.

MPI_ATTR_DELETE(comm, keyval)

| | | |
|---|---|---|
| INOUT | comm | communicator to which attribute is attached (handle) |
| IN | keyval | The key value of the deleted attribute (integer) |

**C binding**

```
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

**F binding**

```
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR
```

## 14.2 Deprecated since MPI-2.2

The entire set of C++ language bindings have been removed. See Chapter 15, Removed Interfaces for more information.

The following function typedefs have been deprecated and are superseded by new names. Other than the typedef names, the function signatures are exactly the same; the names were updated to match conventions of other function typedef names.

| Deprecated Name | New Name |
|---|---|
| MPI_Comm_errhandler_fn | MPI_Comm_errhandler_function |
| MPI_File_errhandler_fn | MPI_File_errhandler_function |
| MPI_Win_errhandler_fn | MPI_Win_errhandler_function |

## 14.3 Deprecated since MPI-3.2

Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.

# Chapter 15

# Removed Interfaces

## 15.1 Removed MPI-1 Bindings

### 15.1.1 Overview

The following MPI-1 bindings were deprecated as of MPI-2 and are removed in MPI-3. They may be provided by an implementation for backwards compatibility, but are not required. Removal of these bindings affects all language-specific definitions thereof. Only the language-neutral bindings are listed when possible.

### 15.1.2 Removed MPI-1 Functions

Table 15.1 shows the removed MPI-1 functions and their replacements.

| Removed | MPI-2 Replacement |
|---|---|
| MPI_ADDRESS | MPI_GET_ADDRESS |
| MPI_ERRHANDLER_CREATE | MPI_COMM_CREATE_ERRHANDLER |
| MPI_ERRHANDLER_GET | MPI_COMM_GET_ERRHANDLER |
| MPI_ERRHANDLER_SET | MPI_COMM_SET_ERRHANDLER |
| MPI_TYPE_EXTENT | MPI_TYPE_GET_EXTENT |
| MPI_TYPE_HINDEXED | MPI_TYPE_CREATE_HINDEXED |
| MPI_TYPE_HVECTOR | MPI_TYPE_CREATE_HVECTOR |
| MPI_TYPE_LB | MPI_TYPE_GET_EXTENT |
| MPI_TYPE_STRUCT | MPI_TYPE_CREATE_STRUCT |
| MPI_TYPE_UB | MPI_TYPE_GET_EXTENT |

Table 15.1: Removed MPI-1 functions and their replacements

### 15.1.3 Removed MPI-1 Datatypes

Table 15.2 shows the removed MPI-1 datatypes and their replacements.

### 15.1.4 Removed MPI-1 Constants

Table 15.3 shows the removed MPI-1 constants. There are no MPI-2 replacements.

| Removed | MPI-2 Replacement |
|---|---|
| MPI_LB | MPI_TYPE_CREATE_RESIZED |
| MPI_UB | MPI_TYPE_CREATE_RESIZED |

Table 15.2: Removed MPI-1 datatypes and their replacements

| Removed MPI-1 Constants |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_COMBINER_HINDEXED_INTEGER |
| MPI_COMBINER_HVECTOR_INTEGER |
| MPI_COMBINER_STRUCT_INTEGER |

Table 15.3: Removed MPI-1 constants

### 15.1.5 Removed MPI-1 Callback Prototypes

Table 15.4 shows the removed MPI-1 callback prototypes and their MPI-2 replacements.

| Removed | MPI-2 Replacement |
|---|---|
| MPI_Handler_function | MPI_Comm_errhandler_function |

Table 15.4: Removed MPI-1 callback prototypes and their replacements

## 15.2 C++ Bindings

The C++ bindings were deprecated as of MPI-2.2. The C++ bindings are removed in MPI-3.0. The namespace is still reserved, however, and bindings may only be provided by an implementation as described in the MPI-2.2 standard.

# Chapter 16

# Backward Incompatibilities

## 16.1 Backward Incompatible since MPI-3.2

The default communicator where errors are raised when not involving a communicator, window, or file was changed from MPI_COMM_WORLD to MPI_COMM_SELF.

# Chapter 17

# Language Bindings

## 17.1 Fortran Support

### 17.1.1 Overview

The Fortran MPI language bindings have been designed to be compatible with the Fortran 90 standard with additional features from Fortran 2003 and Fortran 2008 [40] + TS 29113 [41].

> *Rationale.* Fortran 90 contains numerous features designed to make it a more "modern" language than Fortran 77. It seems natural that MPI should be able to take advantage of these new features with a set of bindings tailored to Fortran 90. In Fortran 2008 + TS 29113, the major new language features used are the `ASYNCHRONOUS` attribute to protect nonblocking MPI operations, and assumed-type and assumed-rank dummy arguments for choice buffer arguments. Further requirements for compiler support are listed in Section 17.1.7. (*End of rationale.*)

MPI defines three methods of Fortran support:

1. **USE mpi_f08:** This method is described in Section 17.1.2. It requires compile-time argument checking with unique MPI handle types and provides techniques to fully solve the optimization problems with nonblocking calls. This is the only Fortran support method that is consistent with the Fortran standard (Fortran 2008 + TS 29113 and later). This method is highly recommended for all MPI applications.

2. **USE mpi:** This method is described in Section 17.1.3 and requires compile-time argument checking. Handles are defined as `INTEGER`. This Fortran support method is inconsistent with the Fortran standard, and its use is therefore not recommended. It exists only for backwards compatibility.

3. **INCLUDE 'mpif.h':** This method is described in Section 17.1.4. The use of the include file `mpif.h` is strongly discouraged starting with MPI-3.0, because this method neither guarantees compile-time argument checking nor provides sufficient techniques to solve the optimization problems with nonblocking calls, and is therefore inconsistent with the Fortran standard. It exists only for backwards compatibility with legacy MPI applications.

Compliant MPI-3 implementations providing a Fortran interface must provide one or both of the following:

- The `USE mpi_f08` Fortran support method.
- The `USE mpi` and `INCLUDE 'mpif.h'` Fortran support methods.

Section 17.1.6 describes restrictions if the compiler does not support all the needed features.

Application subroutines and functions may use either one of the modules or the `mpif.h` include file. An implementation may require the use of one of the modules to prevent type mismatch errors.

> *Advice to users.* Users are advised to utilize one of the MPI modules even if `mpif.h` enforces type checking on a particular system. Using a module provides several potential advantages over using an include file; the `mpi_f08` module offers the most robust and complete Fortran support. (*End of advice to users.*)

In a single application, it must be possible to link together routines which `USE mpi_f08`, `USE mpi`, and `INCLUDE 'mpif.h'`.

The `LOGICAL` compile-time constant MPI_SUBARRAYS_SUPPORTED is set to `.TRUE.` if all buffer choice arguments are defined in explicit interfaces with assumed-type and assumed-rank [41]; otherwise it is set to `.FALSE.`. The `LOGICAL` compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING is set to `.TRUE.` if the `ASYNCHRONOUS` attribute was added to the choice buffer arguments of all nonblocking interfaces **and** the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TS 29113), otherwise it is set to `.FALSE.`. These constants exist for each Fortran support method, but not in the C header file. The values may be different for each Fortran support method. All other constants and the integer values of handles must be the same for each Fortran support method.

Section 17.1.2 through 17.1.4 define the Fortran support methods. The Fortran interfaces of each MPI routine are shorthands. Section 17.1.5 defines the corresponding full interface specification together with the specific procedure names and implications for the profiling interface. Section 17.1.6 the implementation of the MPI routines for different versions of the Fortran standard. Section 17.1.7 summarizes major requirements for valid MPI-3.0 implementations with Fortran support. Section 17.1.8 and Section 17.1.9 describe additional functionality that is part of the Fortran support. MPI_F_SYNC_REG is needed for one of the methods to prevent register optimization problems. A set of functions provides 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. In the context of MPI, parameterized types are Fortran intrinsic types which are specified using `KIND` type parameters. Sections 17.1.10 through 17.1.19 give an overview and details on known problems when using Fortran together with MPI; Section 17.1.20 compares the Fortran problems with those in C.

### 17.1.2 Fortran Support Through the `mpi_f08` Module

An MPI implementation providing a Fortran interface must provide a module named `mpi_f08` that can be used in a Fortran program. Section 17.1.6 describes restrictions if the compiler does not support all the needed features. Within all MPI function specifications, the first of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking for all arguments which are not `TYPE(*)`, with the following exception:

  Only one Fortran interface is defined for functions that are deprecated as of MPI-3.0. This interface must be provided as an explicit interface according to the rules defined for the `mpi` module, see Section 17.1.3.

  *Advice to users.* It is strongly recommended that developers substitute calls to deprecated routines when upgrading from `mpif.h` or the `mpi` module to the `mpi_f08` module. (*End of advice to users.*)

- Define the derived type `MPI_Status`, and define all MPI handles with uniquely named handle types (instead of `INTEGER` handles, as in the `mpi` module). This is reflected in the first Fortran binding in each MPI function definition throughout this document (except for the deprecated routines).
- Overload the operators `.EQ.` and `.NE.` to allow the comparison of these MPI handles with `.EQ.`, `.NE.`, `==` and `/=`.
- Use the `ASYNCHRONOUS` attribute to protect the buffers of nonblocking operations, and set the `LOGICAL` compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to `.TRUE.` if the underlying Fortran compiler supports the `ASYNCHRONOUS` attribute for MPI communication (as part of TS 29113). See Section 17.1.6 for older compiler versions.
- Set the `LOGICAL` compile-time constant MPI_SUBARRAYS_SUPPORTED to `.TRUE.` and declare choice buffers using the Fortran 2008 TS 29113 features assumed-type and assumed-rank, i.e., `TYPE(*), DIMENSION(..)` in all nonblocking, split collective and persistent communication routines, if the underlying Fortran compiler supports it. With this, non-contiguous sub-arrays can be used as buffers in nonblocking routines.

  *Rationale.* In all blocking routines, i.e., if the choice-buffer is not declared as `ASYNCHRONOUS`, the TS 29113 feature is not needed for the support of non-contiguous buffers because the compiler can pass the buffer by in-and-out-copy through a contiguous scratch array. (*End of rationale.*)

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to `.FALSE.` and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the Fortran 2008 TS 29113 assumed-type and assumed-rank notation. In this case, the use of non-contiguous sub-arrays as buffers in nonblocking calls may be invalid. See Section 17.1.6 for details.
- Declare each argument with an `INTENT` of `IN`, `OUT`, or `INOUT` as defined in this standard.

*Rationale.* For these definitions in the mpi_f08 bindings, in most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for OUT and INOUT dummy arguments that allow one of the non-ordinary Fortran constants (see MPI_BOTTOM, etc. in Section 2.5.4) as input, an INTENT is not specified. (*End of rationale.*)

*Advice to users.* If a dummy argument is declared with INTENT(OUT), then the Fortran standard stipulates that the actual argument becomes undefined upon invocation of the MPI routine, i.e., it may be overwritten by some other values, e.g. zeros; according to [40], 12.5.2.4 Ordinary dummy variables, Paragraph 17: "If a dummy argument has INTENT(OUT), the actual argument becomes undefined at the time the association is established, except [...]". For example, if the dummy argument is an assumed-size array and the actual argument is a strided array, the call may be implemented with copy-in and copy-out of the argument. In the case of INTENT(OUT) the copy-in may be suppressed by the optimization and the routine starts execution using an array of undefined values. If the routine stores fewer elements into the dummy argument than is provided in the actual argument, then the remaining locations are overwritten with these undefined values. See also both advices to implementors in Section 17.1.3. (*End of advice to users.*)

- Declare all ierror output arguments as OPTIONAL, except for user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN).

*Rationale.* For user-defined callback functions (e.g., COMM_COPY_ATTR_FUNCTION) and their predefined callbacks (e.g., MPI_COMM_NULL_COPY_FN), the ierror argument is not optional. The MPI library must always call these routines with an actual ierror argument. Therefore, these user-defined functions need not check whether the MPI library calls these routines with or without an actual ierror output argument. (*End of rationale.*)

The MPI Fortran bindings in the mpi_f08 module are designed based on the Fortran 2008 standard [40] together with the Technical Specification "TS 29113 Further Interoperability with C" [41] of the ISO/IEC JTC1/SC22/WG5 (Fortran) working group.

*Rationale.* The features in TS 29113 on further interoperability with C were decided on by ISO/IEC JTC1/SC22/WG5 and designed by PL22.3 (formerly J3) to support a higher level of integration between Fortran-specific features and C than was provided in the Fortran 2008 standard; part of this design is based on requirements from the MPI Forum to support MPI-3.0. According to [41], "an ISO/IEC TS is reviewed after three years in order to decide whether it will be confirmed for a further three years, revised to become an International Standard, or withdrawn. If the ISO/IEC TS is confirmed, it is reviewed again after a further three years, at which time it must either be transformed into an International Standard or be withdrawn."

The TS 29113 contains the following language features that are needed for the MPI bindings in the mpi_f08 module: assumed-type and assumed-rank. It is important that any possible actual argument can be used for such dummy arguments, e.g., scalars, arrays, assumed-shape arrays, assumed-size arrays, allocatable arrays, and with any element type, e.g., REAL, CHARACTER*5, CHARACTER*(*), sequence derived types, or BIND(C) derived types. Especially for backward compatibility reasons, it is important that any possible actual argument in an implicit interface implementation of a choice buffer dummy argument (e.g., with mpif.h without argument-checking) can be used in an implementation with assumed-type and assumed-rank argument in an explicit interface (e.g., with the mpi_f08 module).

A further feature useful for MPI is the extension of the semantics of the ASYNCHRONOUS attribute: In F2003 and F2008, this attribute could be used only to protect buffers of Fortran asynchronous I/O. With TS 29113, this attribute now also covers asynchronous communication occurring within library routines written in C.

The MPI Forum hereby wishes to acknowledge this important effort by the Fortran PL22.3 and WG5 committee. (*End of rationale.*)

### 17.1.3 Fortran Support Through the mpi Module

An MPI implementation providing a Fortran interface must provide a module named mpi that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is provided by this module. This module must:

- Define all named MPI constants
- Declare MPI functions that return a value.
- Provide explicit interfaces according to the Fortran routine interface specifications. This module therefore guarantees compile-time argument checking and allows positional and keyword-based argument lists. If an implementation is paired with a compiler that either does not support TYPE(*), DIMENSION(..) from TS 29113, or is otherwise unable to ignore the types of choice buffers, then the implementation must provide explicit interfaces only for MPI routines with no choice buffer arguments. See Section 17.1.6 for more details.
- Define all MPI handles as type INTEGER.
- Define the derived type MPI_Status and all named handle types that are used in the mpi_f08 module. For these named handle types, overload the operators .EQ. and .NE. to allow handle comparison via the .EQ., .NE., == and /= operators.

  *Rationale.* They are needed only when the application converts old-style INTEGER handles into new-style handles with a named type. (*End of rationale.*)
- A high quality MPI implementation may enhance the interface by using the ASYNCHRONOUS attribute in the same way as in the mpi_f08 module if it is supported by the underlying compiler.
- Set the LOGICAL compile-time constant MPI_ASYNC_PROTECTS_NONBLOCKING to .TRUE. if the ASYNCHRONOUS attribute is used in all nonblocking interfaces **and** the underlying Fortran compiler supports the ASYNCHRONOUS attribute for MPI communication (as part of TS 29113), otherwise to .FALSE..

*Advice to users.* For an MPI implementation that fully supports nonblocking calls with the ASYNCHRONOUS attribute for choice buffers, an existing MPI-2.2 application may fail to compile even if it compiled and executed with expected results with an MPI-2.2 implementation. One reason may be that the application uses "contiguous" but not "simply contiguous" ASYNCHRONOUS arrays as actual arguments for choice buffers of nonblocking routines, e.g., by using subscript triplets with stride one or specifying (1:n) for a whole dimension instead of using (:). This should be fixed to fulfill the Fortran constraints for ASYNCHRONOUS dummy arguments. This is not considered a violation of backward compatibility because existing applications can not use the ASYNCHRONOUS attribute to protect nonblocking calls. Another reason may be that the application does not conform either to MPI-2.2, or to MPI-3.0, or to the Fortran standard, typically because the program forces the compiler to perform copy-in/out for a choice buffer argument in a nonblocking MPI call. This is also not a violation of backward compatibility because the application itself is non-conforming. See Section 17.1.12 for more details. (*End of advice to users.*)

- A high quality MPI implementation may enhance the interface by using TYPE(*), DIMENSION(..) choice buffer dummy arguments instead of using non-standardized extensions such as !$PRAGMA IGNORE_TKR or a set of overloaded functions as described by M. Hennecke in [28], if the compiler supports this TS 29113 language feature. See Section 17.1.6 for further details.

- Set the LOGICAL compile-time constant MPI_SUBARRAYS_SUPPORTED to .TRUE. if all choice buffer arguments in all nonblocking, split collective and persistent communication routines are declared with TYPE(*), DIMENSION(..), otherwise set it to .FALSE.. When MPI_SUBARRAYS_SUPPORTED is defined as .TRUE., non-contiguous sub-arrays can be used as buffers in nonblocking routines.

- Set the MPI_SUBARRAYS_SUPPORTED compile-time constant to .FALSE. and declare choice buffers with a compiler-dependent mechanism that overrides type checking if the underlying Fortran compiler does not support the TS 29113 assumed-type and assumed-rank features. In this case, the use of non-contiguous sub-arrays in nonblocking calls may be disallowed. See Section 17.1.6 for details.

An MPI implementation may provide other features in the mpi module that enhance the usability of MPI while maintaining adherence to the standard. For example, it may provide INTENT information in these interface blocks.

*Advice to implementors.* The appropriate INTENT may be different from what is given in the MPI language-neutral bindings. Implementations must choose INTENT so that the function adheres to the MPI standard, e.g., by defining the INTENT as provided in the mpi_f08 bindings. (*End of advice to implementors.*)

*Rationale.* The intent given by the MPI generic interface is not precisely defined and does not in all cases correspond to the correct Fortran INTENT. For instance, receiving into a buffer specified by a datatype with absolute addresses may require associating MPI_BOTTOM with a dummy OUT argument. Moreover, "constants" such 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 was changed in several places in MPI-2. For instance, MPI_IN_PLACE changes the intent of an OUT argument to be INOUT. (*End of rationale.*)

*Advice to implementors.* The Fortran 2008 standard illustrates in its Note 5.17 that "INTENT(OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If an argument should retain its value rather than being redefined, INTENT(INOUT) should be used rather than INTENT(OUT), even if there is no explicit reference to the value of the dummy argument. Furthermore, INTENT(INOUT) is not equivalent to omitting the INTENT attribute, because INTENT(INOUT) always requires that the associated actual argument is definable." Applications that include `mpif.h` may not expect that `INTENT(OUT)` is used. In particular, output array arguments are expected to keep their content as long as the MPI routine does not modify them. To keep this behavior, it is recommended that implementations not use `INTENT(OUT)` in the `mpi` module and the `mpif.h` include file, even though `INTENT(OUT)` is specified in an interface description of the `mpi_f08` module. (*End of advice to implementors.*)

### 17.1.4 Fortran Support Through the `mpif.h` Include File

The use of the `mpif.h` include file is strongly discouraged and may be deprecated in a future version of MPI.

An MPI implementation providing a Fortran interface must provide an include file named `mpif.h` that can be used in a Fortran program. Within all MPI function specifications, the second of the set of two Fortran routine interface specifications is supported by this include file. This include file must:

- Define all named MPI constants.
- Declare MPI functions that return a value.
- Define all handles as `INTEGER`.
- Be valid and equivalent for both fixed and free source form.

For each MPI routine, an implementation can choose to use an implicit or explicit interface for the second Fortran binding (in deprecated routines, the first one may be omitted).

- Set the `LOGICAL` compile-time constants MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING according to the same rules as for the `mpi` module. In the case of implicit interfaces for choice buffer or nonblocking routines, the constants must be set to `.FALSE.`.

*Advice to users.* Instead of using `mpif.h`, the use of the `mpi_f08` or `mpi` module is strongly encouraged for the following reasons:

- Most `mpif.h` implementations do not include compile-time argument checking.
- Therefore, many bugs in MPI applications remain undetected at compile-time, such as:
  - Missing ierror as last argument in most Fortran bindings.

  - Declaration of a status as an INTEGER variable instead of an INTEGER array with size MPI_STATUS_SIZE.
  - Incorrect argument positions; e.g., interchanging the count and datatype arguments.
  - Passing incorrect MPI handles; e.g., passing a datatype instead of a communicator.

- The migration from mpif.h to the mpi module should be relatively straightforward (i.e., substituting include 'mpif.h' after an implicit statement by use mpi before that implicit statement) as long as the application syntax is correct.
- Migrating portable and correctly written applications to the mpi module is not expected to be difficult. No compile or runtime problems should occur because an mpif.h include file was always allowed to provide explicit Fortran interfaces.

(*End of advice to users.*)

*Rationale.* With MPI-3.0, the mpif.h include file was not deprecated in order to retain strong backward compatibility. Internally, mpif.h and the mpi module may be implemented so that essentialy the same library implementation of the MPI routines can be used. (*End of rationale.*)

### 17.1.5 Interface Specifications, Procedure Names, and the Profiling Interface

The Fortran interface specification of each MPI routine specifies the routine name that must be called by the application program, and the names and types of the dummy arguments together with additional attributes. The Fortran standard allows a given Fortran interface to be implemented with several methods, e.g., within or outside of a module, with or without BIND(C), or the buffers with or without TS 29113. Such implementation decisions imply different binary interfaces and different specific procedure names. The requirements for several implementation schemes together with the rules for the specific procedure names and its implications for the profiling interface are specified within this section, but not the implementation details.

*Rationale.* This section was introduced in MPI-3.0 on Sep. 21, 2012. The major goals for implementing the three Fortran support methods have been:

- Portable implementation of the wrappers from the MPI Fortran interfaces to the MPI routines in C.
- Binary backward compatible implementation path when switching MPI_SUBARRAYS_SUPPORTED from .FALSE. to .TRUE..
- The Fortran PMPI interface need not be backward compatible, but a method must be included that a tools layer can use to examine the MPI library about the specific procedure names and interfaces used.
- No performance drawbacks.
- Consistency between all three Fortran support methods.
- Consistent with Fortran 2008 + TS 29113.

| No. | Specific procedure name | Calling convention |
|---|---|---|
| 1A | MPI_Isend_f08 | Fortran interface and arguments, as in Annex A.3, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like `!$PRAGMA IGNORE_TKR`, which provides a call-by-reference argument without type, kind, and dimension checking. |
| 1B | MPI_Isend_f08ts | Fortran interface and arguments, as in Annex A.3, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with `TYPE(*), DIMENSION(..)`. |
| 2A | MPI_ISEND | Fortran interface and arguments, as in Annex A.4, except that in routines with a choice buffer dummy argument, this dummy argument is implemented with non-standard extensions like `!$PRAGMA IGNORE_TKR`, which provides a call-by-reference argument without type, kind, and dimension checking. |
| 2B | MPI_ISEND_FTS | Fortran interface and arguments, as in Annex A.4, but only for routines with one or more choice buffer dummy arguments; these dummy arguments are implemented with `TYPE(*), DIMENSION(..)`. |

Table 17.1: Specific Fortran procedure names and related calling conventions. MPI_ISEND is used as an example. For routines without choice buffers, only 1A and 2A apply.

The design expected that all dummy arguments in the MPI Fortran interfaces are interoperable with C according to Fortran 2008 + TS 29113. This expectation was not fulfilled. The `LOGICAL` arguments are not interoperable with C, mainly because the internal representations for `.FALSE.` and `.TRUE.` are compiler dependent. The provided interface was mainly based on `BIND(C)` interfaces and therefore inconsistent with Fortran. To be consistent with Fortran, the `BIND(C)` had to be removed from the callback procedure interfaces and the predefined callbacks, e.g., MPI_COMM_DUP_FN. Non-`BIND(C)` procedures are also not interoperable with C, and therefore the `BIND(C)` had to be removed from all routines with `PROCEDURE` arguments, e.g., from MPI_OP_CREATE.

Therefore, this section was rewritten as an erratum to MPI-3.0. (*End of rationale.*)

A Fortran call to an MPI routine shall result in a call to a procedure with one of the specific procedure names and calling conventions, as described in Table 17.1. Case is not significant in the names.

Note that for the deprecated routines in Section 14.1, which are reported only in Annex A.4, scheme 2A is utilized in the `mpi` module and `mpif.h`, and also in the `mpi_f08` module.

To set MPI_SUBARRAYS_SUPPORTED to `.TRUE.` within a Fortran support method, it is required that all non-blocking and split-collective routines with buffer arguments are implemented according to 1B and 2B, i.e., with MPI_Xxxx_f08ts in the mpi_f08 module, and with MPI_XXXX_FTS in the mpi module and the mpif.h include file.

The mpi and mpi_f08 modules and the mpif.h include file will each correspond to exactly one implementation scheme from Table 17.1. However, the MPI library may contain multiple implementation schemes from Table 17.1.

*Advice to implementors.* This may be desirable for backwards binary compatibility in the scope of a single MPI implementation, for example. (*End of advice to implementors.*)

*Rationale.* After a compiler provides the facilities from TS 29113, i.e., TYPE(*), DIMENSION(..), it is possible to change the bindings within a Fortran support method to support subarrays without recompiling the complete application provided that the previous interfaces with their specific procedure names are still included in the library. Of course, only recompiled routines can benefit from the added facilities. There is no binary compatibility conflict because each interface uses its own specific procedure names and all interfaces use the same constants (except the value of MPI_SUBARRAYS_SUPPORTED and MPI_ASYNC_PROTECTS_NONBLOCKING) and type definitions. After a compiler also ensures that buffer arguments of nonblocking MPI operations can be protected through the ASYNCHRONOUS attribute, and the procedure declarations in the mpi_f08 and mpi module and the mpif.h include file declare choice buffers with the ASYNCHRONOUS attribute, then the value of MPI_ASYNC_PROTECTS_NONBLOCKING can be switched to .TRUE. in the module definition and include file. (*End of rationale.*)

*Advice to users.* Partial recompilation of user applications when upgrading MPI implementations is a highly complex and subtle topic. Users are strongly advised to consult their MPI implementation's documentation to see exactly what is — and what is not — supported. (*End of advice to users.*)

Within the mpi_f08 and mpi modules and mpif.h, for all MPI procedures, a second procedure with the same calling conventions shall be supplied, except that the name is modified by prefixing with the letter "P", e.g., PMPI_Isend. The specific procedure names for these PMPI_Xxxx procedures must be different from the specific procedure names for the MPI_Xxxx procedures and are not specified by this standard.

A user-written or middleware profiling routine should provide the same specific Fortran procedure names and calling conventions, and therefore can interpose itself as the MPI library routine. The profiling routine can internally call the matching PMPI routine with any of its existing bindings, except for routines that have callback routine dummy arguments, choice buffer arguments, or that are attribute caching routines ( MPI_{COMM|WIN|TYPE}_{SET|GET}_ATTR). In this case, the profiling software should invoke the corresponding PMPI routine using the same Fortran support method as used in the calling application program, because the C, mpi_f08 and mpi callback prototypes are different or the meaning of the choice buffer or attribute_val arguments are different.

*Advice to users.* Although for each support method and MPI routine (e.g., MPI_ISEND in mpi_f08), multiple routines may need to be provided to intercept the specific procedures in the MPI library (e.g., MPI_Isend_f08 and MPI_Isend_f08ts), each profiling routine itself uses only one support method (e.g., mpi_f08) and calls the real MPI routine through the one PMPI routine defined in this support method (i.e., PMPI_Isend in this example). (*End of advice to users.*)

*Advice to implementors.* If all of the following conditions are fulfilled:

- the handles in the mpi_f08 module occupy one Fortran numerical storage unit (same as an INTEGER handle),
- the internal argument passing mechanism used to pass an actual ierror argument to a non-optional ierror dummy argument is binary compatible to passing an actual ierror argument to an ierror dummy argument that is declared as OPTIONAL,
- the internal argument passing mechanism for ASYNCHRONOUS and non-ASYNCHRONOUS arguments is the same,
- the internal routine call mechanism is the same for the Fortran and the C compilers for which the MPI library is compiled,
- the compiler does not provide TS 29113,

then the implementor may use the same internal routine implementations for all Fortran support methods but with several different specific procedure names. If the accompanying Fortran compiler supports TS 29113, then the new routines are needed only for routines with choice buffer arguments. (*End of advice to implementors.*)

*Advice to implementors.* In the Fortran support method mpif.h, compile-time argument checking can be also implemented for all routines. For mpif.h, the argument names are not specified through the MPI standard, i.e., only positional argument lists are defined, and not key-word based lists. Due to the rule that mpif.h must be valid for fixed and free source form, the subroutine declaration is restricted to one line with 72 characters. To keep the argument lists short, each argument name can be shortened to a minimum of one character. With this, the two longest subroutine declaration statements are

```
SUBROUTINE PMPI_Dist_graph_create_adjacent(a,b,c,d,e,f,g,h,i,j,k)
SUBROUTINE PMPI_Rget_accumulate(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 71 and 66 characters. With buffers implemented with TS 29113, the specific procedure names have an additional postfix. The longest of such interface definitions is

```
INTERFACE  PMPI_Rget_accumulate
SUBROUTINE PMPI_Rget_accumulate_fts(a,b,c,d,e,f,g,h,i,j,k,l,m,n)
```

with 70 characters. In principle, continuation lines would be possible in mpif.h (spaces in columns 73–131, & in column 132, and in column 6 of the continuation line) but this would not be valid if the source line length is extended with a compiler flag to 132 characters. Column 133 is also not available for the continuation character because lines longer than 132 characters are invalid with some compilers by default.

The longest specific procedure names are PMPI_Dist_graph_create_adjacent_f08 and PMPI_File_write_ordered_begin_f08ts both with 35 characters in the mpi_f08 module.

For example, the interface specifications together with the specific procedure names can be implemented with

```
MODULE mpi_f08
  TYPE, BIND(C) :: MPI_Comm
    INTEGER :: MPI_VAL
  END TYPE MPI_Comm
  ...
  INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
    SUBROUTINE MPI_Comm_rank_f08(comm, rank, ierror)
      IMPORT :: MPI_Comm
      TYPE(MPI_Comm),      INTENT(IN)  :: comm
      INTEGER,             INTENT(OUT) :: rank
      INTEGER, OPTIONAL,   INTENT(OUT) :: ierror
    END SUBROUTINE
  END INTERFACE
END MODULE mpi_f08

MODULE mpi
  INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
    SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
      INTEGER, INTENT(IN)  :: comm   ! The INTENT may be added although
      INTEGER, INTENT(OUT) :: rank   ! it is not defined in the
      INTEGER, INTENT(OUT) :: ierror ! official routine definition.
    END SUBROUTINE
  END INTERFACE
END MODULE mpi
```

And if interfaces are provided in `mpif.h`, they might look like this (outside of any module and in fixed source format):

```
!23456789012345678901234567890123456789012345678901234567890123456789012
      INTERFACE MPI_Comm_rank  ! (as defined in Chapter 6)
       SUBROUTINE MPI_Comm_rank(comm, rank, ierror)
        INTEGER, INTENT(IN)  :: comm   ! The argument names may be
        INTEGER, INTENT(OUT) :: rank   ! shortened so that the
        INTEGER, INTENT(OUT) :: ierror ! subroutine line fits to the
       END SUBROUTINE                  ! maximum of 72 characters.
      END INTERFACE
```

(*End of advice to implementors.*)

*Advice to users.* The following is an example of how a user-written or middleware profiling routine can be implemented:

```
SUBROUTINE MPI_Isend_f08ts(buf,count,datatype,dest,tag,comm,request,ierror)
  USE :: mpi_f08, my_noname => MPI_Isend_f08ts
  TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
  INTEGER,             INTENT(IN)      :: count, dest, tag
  TYPE(MPI_Datatype), INTENT(IN)       :: datatype
  TYPE(MPI_Comm),      INTENT(IN)      :: comm
  TYPE(MPI_Request),  INTENT(OUT)      :: request
  INTEGER, OPTIONAL,  INTENT(OUT)      :: ierror
    ! ... some code for the begin of profiling
  call PMPI_Isend (buf, count, datatype, dest, tag, comm, request, ierror)
    ! ... some code for the end of profiling
END SUBROUTINE MPI_Isend_f08ts
```

Note that this routine is used to intercept the existing specific procedure name MPI_Isend_f08ts in the MPI library. This routine must not be part of a module. This routine itself calls PMPI_Isend. The USE of the mpi_f08 module is needed for definitions of handle types and the interface for PMPI_Isend. However, this module also contains an interface definition for the specific procedure name MPI_Isend_f08ts that conflicts with the definition of this profiling routine (i.e., the name is doubly defined). Therefore, the USE here specifically excludes the interface from the module by renaming the unused routine name in the mpi_f08 module into "my_noname" in the scope of this routine. (*End of advice to users.*)

*Advice to users.* The PMPI interface allows intercepting MPI routines. For example, an additional MPI_ISEND profiling wrapper can be provided that is called by the application and internally calls PMPI_ISEND. There are two typical use cases: a profiling layer that is developed independently from the application and the MPI library, and profiling routines that are part of the application and have access to the application data. With MPI-3.0, new Fortran interfaces and implementation schemes were introduced that have several implications on how Fortran MPI routines are internally implemented and optimized. For profiling layers, these schemes imply that several internal interfaces with different specific procedure names may need to be intercepted, as shown in the example code above. Therefore, for wrapper routines that are part of a Fortran application, it may be more convenient to make the name shift within the application, i.e., to substitute the call to the MPI routine (e.g., MPI_ISEND) by a call to a user-written profiling wrapper with a new name (e.g., X_MPI_ISEND) and to call the Fortran MPI_ISEND from this wrapper, instead of using the PMPI interface. (*End of advice to users.*)

*Advice to implementors.* An implementation that provides a Fortran interface must provide a combination of MPI library and module or include file that uses the specific procedure names as described in Table 17.1 so that the MPI Fortran routines are interceptable as described above. (*End of advice to implementors.*)

### 17.1.6 MPI for Different Fortran Standard Versions

This section describes which Fortran interface functionality can be provided for different versions of the Fortran standard.

- *For Fortran 77* with some extensions:
  - MPI identifiers may be up to 30 characters (31 with the profiling interface).
  - MPI identifiers may contain underscores after the first character.
  - An MPI subroutine with a choice argument may be called with different argument types.
  - Although not required by the MPI standard, the INCLUDE statement should be available for including mpif.h into the user application source code.

  Only MPI-1.1, MPI-1.2, and MPI-1.3 can be implemented. The use of absolute addresses from MPI_ADDRESS and MPI_BOTTOM may cause problems if an address does not fit into the memory space provided by an INTEGER. (In MPI-2.0 this problem is solved with MPI_GET_ADDRESS, but not for Fortran 77.)

- *For Fortran 90:*
  The major additional features that are needed from Fortran 90 are:
  - The MODULE and INTERFACE concept.
  - The KIND= and SELECTED_..._KIND concept.
  - Fortran derived TYPEs and the SEQUENCE attribute.
  - The OPTIONAL attribute for dummy arguments.
  - Cray pointers, which are a non-standard compiler extension, are needed for the use of MPI_ALLOC_MEM.

  With these features, MPI-1.1 – MPI-2.2 can be implemented without restrictions. MPI-3.0 can be implemented with some restrictions. The Fortran support methods are abbreviated with S1 = the mpi_f08 module, S2 = the mpi module, and S3 = the mpif.f include file. If not stated otherwise, restrictions exist for each method which prevent implementing the complete semantics of MPI-3.0.

  - MPI_SUBARRAYS_SUPPORTED equals .FALSE., i.e., subscript triplets and non-contiguous subarrays cannot be used as buffers in nonblocking routines, RMA, or split-collective I/O.
  - S1, S2, and S3 can be implemented, but for S1, only a preliminary implementation is possible.
  - In this preliminary interface of S1, the following changes are necessary:
    * TYPE(*), DIMENSION(..) is substituted by non-standardized extensions like !$PRAGMA IGNORE_TKR.
    * The ASYNCHRONOUS attribute is omitted.
    * PROCEDURE(...) callback declarations are substituted by EXTERNAL.
  - The specific procedure names are specified in Section 17.1.5.
  - Due to the rules specified in Section 17.1.5, choice buffer declarations should be implemented only with non-standardized extensions like !$PRAGMA IGNORE_TKR (as long as F2008+TS 29113 is not available).

    In S2 and S3: Without such extensions, routines with choice buffers should be provided with an implicit interface, instead of overloading with a different MPI function for each possible buffer type (as mentioned in Section 17.1.11). Such overloading would also imply restrictions for passing Fortran derived types as choice buffer, see also Section 17.1.15.

    Only in S1: The implicit interfaces for routines with choice buffer arguments imply that the ierror argument cannot be defined as OPTIONAL. For this reason, it is recommended not to provide the mpi_f08 module if such an extension is not available.
  - The ASYNCHRONOUS attribute can **not** be used in applications to protect buffers in nonblocking MPI calls (S1–S3).
  - The TYPE(C_PTR) binding of the MPI_ALLOC_MEM and MPI_WIN_ALLOCATE routines is not available.

  - In S1 and S2, the definition of the handle types (e.g., `TYPE(MPI_Comm)` and the status type `TYPE(MPI_Status)` must be modified: The `SEQUENCE` attribute must be used instead of `BIND(C)` (which is not available in Fortran 90/95). This restriction implies that the application must be fully recompiled if one switches to an MPI library for Fortran 2003 and later because the internal memory size of the handles may have changed. For this reason, an implementor may choose not to provide the `mpi_f08` module for Fortran 90 compilers. In this case, the `mpi_f08` handle types and all routines, constants and types related to `TYPE(MPI_Status)` (see Section 17.2.5) are also not available in the `mpi` module and `mpif.h`.

- *For Fortran 95:*
  The quality of the MPI interface and the restrictions are the same as with Fortran 90.

- *For Fortran 2003:*
  The major features that are needed from Fortran 2003 are:

  - Interoperability with C, i.e.,
    * `BIND(C)` derived types.
    * The `ISO_C_BINDING` intrinsic type `C_PTR` and routine `C_F_POINTER`.
  - The ability to define an `ABSTRACT INTERFACE` and to use it for `PROCEDURE` dummy arguments.
  - The ability to overload the operators `.EQ.` and `.NE.` to allow the comparison of derived types (used in MPI-3.0 for MPI handles).
  - The `ASYNCHRONOUS` attribute is available to protect Fortran asynchronous I/O. This feature is not yet used by MPI, but it is the basis for the enhancement for MPI communication in the TS 29113.

  With these features (but still without the features of TS 29113), MPI-1.1 – MPI-2.2 can be implemented without restrictions, but with one enhancement:

  - The user application can use `TYPE(C_PTR)` together with MPI_ALLOC_MEM as long as MPI_ALLOC_MEM is defined with an implicit interface because a `C_PTR` and an `INTEGER(KIND=MPI_ADDRESS_KIND)` argument must both map to a `void *` argument.

  MPI-3.0 can be implemented with the following restrictions:

  - MPI_SUBARRAYS_SUPPORTED equals `.FALSE.`.
  - For S1, only a preliminary implementation is possible. The following changes are necessary:
    * `TYPE(*), DIMENSION(..)` is substituted by non-standardized extensions like `!$PRAGMA IGNORE_TKR`.
  - The specific procedure names are specified in Section 17.1.5.
  - With S1, the `ASYNCHRONOUS` is required as specified in the second Fortran interfaces. With S2 and S3 the implementation can also add this attribute if explicit interfaces are used.

   - The ASYNCHRONOUS Fortran attribute can be used in applications to *try to* protect buffers in nonblocking MPI calls, but the protection can work only if the compiler is able to protect asynchronous Fortran I/O and makes no difference between such asynchronous Fortran I/O and MPI communication.
   - The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used only for Fortran types that are C compatible.
   - The same restriction as for Fortran 90 applies if non-standardized extensions like !$PRAGMA IGNORE_TKR are not available.

- *For Fortran 2008 + TS 29113 and later* and
  *For Fortran 2003 + TS 29113:*
  The major feature that are needed from TS 29113 are:
   - TYPE(*), DIMENSION(..) is available.
   - The ASYNCHRONOUS attribute is extended to protect also nonblocking MPI communication.
   - The array dummy argument of the ISO_C_BINDING intrinsic C_F_POINTER is not restricted to Fortran types for which a corresponding type in C exists.

  Using these features, MPI-3.0 can be implemented without any restrictions.
   - With S1, MPI_SUBARRAYS_SUPPORTED equals .TRUE.. The ASYNCHRONOUS attribute can be used to protect buffers in nonblocking MPI calls. The TYPE(C_PTR) binding of the MPI_ALLOC_MEM, MPI_WIN_ALLOCATE, MPI_WIN_ALLOCATE_SHARED, and MPI_WIN_SHARED_QUERY routines can be used for any Fortran type.
   - With S2 and S3, the value of MPI_SUBARRAYS_SUPPORTED is implementation dependent. A high quality implementation will also provide MPI_SUBARRAYS_SUPPORTED==.TRUE. and will use the ASYNCHRONOUS attribute in the same way as in S1.
   - If non-standardized extensions like !$PRAGMA IGNORE_TKR are not available then S2 must be implemented with TYPE(*), DIMENSION(..).

*Advice to implementors.* If MPI_SUBARRAYS_SUPPORTED==.FALSE., the choice argument may be implemented with an explicit interface using compiler directives, for example:

```
INTERFACE
  SUBROUTINE MPI_...(buf, ...)
    !DEC$ ATTRIBUTES NO_ARG_CHECK :: buf
    !$PRAGMA IGNORE_TKR buf
    !DIR$ IGNORE_TKR buf
    !IBM* IGNORE_TKR buf
    REAL, DIMENSION(*) :: buf
    ... ! declarations of the other arguments
  END SUBROUTINE
END INTERFACE
```

(*End of advice to implementors.*)

### 17.1.7 Requirements on Fortran Compilers

MPI-3.0 (and later) compliant Fortran bindings are not only a property of the MPI library itself, but rather a property of an MPI library together with the Fortran compiler suite for which it is compiled.

> *Advice to users.* Users must take appropriate steps to ensure that proper options are specified to compilers. MPI libraries must document these options. Some MPI libraries are shipped together with special compilation scripts (e.g., `mpif90`, `mpicc`) that set these options automatically. (*End of advice to users.*)

An MPI library together with the Fortran compiler suite is only compliant with MPI-3.0 (and later), as referred by MPI_GET_VERSION, if all the solutions described in Sections 17.1.11 through 17.1.19 work correctly. Based on this rule, major requirements for all three Fortran support methods (i.e., the `mpi_f08` and `mpi` modules, and `mpif.h`) are:

- The language features assumed-type and assumed-rank from Fortran 2008 TS 29113 [41] are available. This is required only for `mpi_f08`. As long as this requirement is not supported by the compiler, it is valid to build an MPI library that implements the `mpi_f08` module with MPI_SUBARRAYS_SUPPORTED set to `.FALSE.`.
- "Simply contiguous" arrays and scalars must be passed to choice buffer dummy arguments of nonblocking routines with call by reference. This is needed only if one of the support methods does not use the `ASYNCHRONOUS` attribute. See Section 17.1.12 for more details.
- `SEQUENCE` and `BIND(C)` derived types are valid as actual arguments passed to choice buffer dummy arguments, and, in the case of MPI_SUBARRAYS_SUPPORTED== `.FALSE.`, they are passed with call by reference, and passed by descriptor in the case of `.TRUE.`.
- All actual arguments that are allowed for a dummy argument in an implicitly defined and separately compiled Fortran routine with the given compiler (e.g., `CHARACTER(LEN=*)` strings and array of strings) must also be valid for choice buffer dummy arguments with all Fortran support methods.
- The array dummy argument of the `ISO_C_BINDING` intrinsic module procedure `C_F_POINTER` is not restricted to Fortran types for which a corresponding type in C exists.
- The Fortran compiler shall not provide `TYPE(*)` unless the `ASYNCHRONOUS` attribute protects MPI communication as described in TS 29113. Specifically, the TS 29113 must be implemented as a whole.

The following rules are required at least as long as the compiler does not provide the extension of the `ASYNCHRONOUS` attribute as part of TS 29113 and there still exists a Fortran support method with MPI_ASYNC_PROTECTS_NONBLOCKING==`.FALSE.`. Observation of these rules by the MPI application developer is especially recomended for backward compatibility of existing applications that use the `mpi` module or the `mpif.h` include file. The rules are as follows:

- Separately compiled empty Fortran routines with implicit interfaces and separately compiled empty C routines with BIND(C) Fortran interfaces (e.g., MPI_F_SYNC_REG on page 661 and Section 17.1.8, and DD on page 662) solve the problems described in Section 17.1.17.
- The problems with temporary data movement (described in detail in Section 17.1.18) are solved as long as the application uses different sets of variables for the nonblocking communication (or nonblocking or split collective I/O) and the computation when overlapping communication and computation.
- Problems caused by automatic and permanent data movement (e.g., within a garbage collection, see Section 17.1.19) are resolved **without** any further requirements on the application program, neither on the usage of the buffers, nor on the declaration of application routines that are involved in invoking MPI procedures.

All of these rules are valid for the `mpi_f08` and `mpi` modules and independently of whether `mpif.h` uses explicit interfaces.

*Advice to implementors.* Some of these rules are already part of the Fortran 2003 standard, some of these requirements require the Fortran TS 29113 [41], and some of these requirements for MPI-3.0 are beyond the scope of TS 29113. (*End of advice to implementors.*)

### 17.1.8 Additional Support for Fortran Register-Memory-Synchronization

As described in Section 17.1.17, a dummy call may be necessary to tell the compiler that registers are to be flushed for a given buffer or that accesses to a buffer may not be moved across a given point in the execution sequence. Only a Fortran binding exists for this call.

MPI_F_SYNC_REG(buf)

| | | |
|---|---|---|
| INOUT | buf | initial address of buffer (choice) |

**F08 binding**

```
MPI_F_sync_reg(buf)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf
```

**F binding**

```
MPI_F_SYNC_REG(BUF)
    <type> BUF(*)
```

This routine has no executable statements. It must be compiled in the MPI library in such a manner that a Fortran compiler cannot detect in the module that the routine has an empty body. It is used only to force the compiler to flush a cached register value of a variable or buffer back to memory (when necessary), or to invalidate the register value.

*Rationale.* This function is not available in other languages because it would not be useful. This routine has no ierror return argument because there is no operation that can fail. (*End of rationale.*)

*Advice to implementors.* This routine can be bound to a C routine to minimize the risk that the Fortran compiler can learn that this routine is empty (and that the call to this routine can be removed as part of an optimization). However, it is explicitly allowed to implement this routine within the `mpi_f08` module according to the definition for the `mpi` module or `mpif.h` to circumvent the overhead of building the internal dope vector to handle the assumed-type, assumed-rank argument. (*End of advice to implementors.*)

*Rationale.* This routine is not defined with `TYPE(*), DIMENSION(*)`, i.e., assumed size instead of assumed rank, because this would restrict the usability to "simply contiguous" arrays and would require overloading with another interface for scalar arguments. (*End of rationale.*)

*Advice to users.* If only a part of an array (e.g., defined by a subscript triplet) is used in a nonblocking routine, it is recommended to pass the whole array to MPI_F_SYNC_REG anyway to minimize the overhead of this no-operation call. Note that this routine need not be called if MPI_ASYNC_PROTECTS_NONBLOCKING is `.TRUE.` and the application fully uses the facilities of `ASYNCHRONOUS` arrays. (*End of advice to users.*)

### 17.1.9 Additional Support for Fortran Numeric Intrinsic Types

MPI provides a small number of named datatypes that correspond to named intrinsic types supported by C and Fortran. These include MPI_INTEGER, MPI_REAL, MPI_INT, MPI_DOUBLE, etc., as well as the optional types MPI_REAL4, MPI_REAL8, etc. There is a one-to-one correspondence between language declarations and MPI types.

Fortran (starting with Fortran 90) provides so-called `KIND`-parameterized types. These types are declared using an intrinsic type (one of `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, and `CHARACTER`) with an optional integer `KIND` parameter that selects from among one or more variants. The specific meaning of different `KIND` values themselves are implementation dependent and not specified by the language. Fortran provides the `KIND` selection functions `selected_real_kind` for `REAL` and `COMPLEX` types, and `selected_int_kind` for `INTEGER` types that allow users to declare variables with a minimum precision or number of digits. These functions provide a portable way to declare `KIND`-parameterized `REAL`, `COMPLEX`, and `INTEGER` variables in Fortran. This scheme is backward compatible with Fortran 77. `REAL` and `INTEGER` Fortran variables have a default `KIND` if none is specified. Fortran `DOUBLE PRECISION` variables are of intrinsic type `REAL` with a non-default `KIND`. The following two declarations are equivalent:

```
double precision x
real(KIND(0.0d0)) x
```

MPI provides two orthogonal methods for handling communication buffers of numeric intrinsic types. The first method (see the following section) 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 (see "Support for size-specific MPI Datatypes" on page 644) gives the user complete control over communication by exposing machine representations.

### Parameterized Datatypes with Specified Precision and Exponent Range

MPI provides named datatypes corresponding to standard Fortran 77 numeric types: MPI_INTEGER, MPI_COMPLEX, MPI_REAL, MPI_DOUBLE_PRECISION and MPI_DOUBLE_COMPLEX. MPI automatically selects the correct data size and provides representation conversion in heterogeneous environments. The mechanism described in this section extends this model to support portable parameterized numeric types.

The model for supporting portable parameterized types is as follows. Real variables are declared (perhaps indirectly) using `selected_real_kind(p, r)` to determine the `KIND` parameter, where `p` is decimal digits of precision and `r` is an exponent range. Implicitly MPI maintains a two-dimensional array of predefined MPI datatypes `D(p, r)`. `D(p, r)` is defined for each value of `(p, r)` supported by the compiler, including pairs for which one value is unspecified. Attempting to access an element of the array with an index `(p, r)` not supported by the compiler is erroneous. MPI implicitly maintains a similar array of `COMPLEX` datatypes. For integers, there is a similar implicit array related to `selected_int_kind` and indexed by the requested number of digits `r`. Note that the predefined datatypes contained in these implicit arrays are not the same as the named MPI datatypes MPI_REAL, etc., but a new set.

*Advice to implementors.* The above description is for explanatory purposes only. It is not expected that implementations will have such internal arrays. (*End of advice to implementors.*)

*Advice to users.* `selected_real_kind()` maps a large number of `(p,r)` pairs to a much smaller number of `KIND` parameters supported by the compiler. `KIND` parameters are not specified by the language and are not portable. From the language point of view intrinsic types of the same base type and `KIND` parameter are of the same type. In order to allow interoperability in a heterogeneous environment, MPI is more stringent. The corresponding MPI datatypes match if and only if they have the same `(p,r)` value (`REAL` and `COMPLEX`) or `r` value (`INTEGER`). Thus MPI has many more datatypes than there are fundamental language types. (*End of advice to users.*)

MPI_TYPE_CREATE_F90_REAL(p, r, newtype)

| | | |
|---|---|---|
| IN | p | precision, in decimal digits (integer) |
| IN | r | decimal exponent range (integer) |
| OUT | newtype | the requested MPI datatype (handle) |

**C binding**

```
int MPI_Type_create_f90_real(int p, int r, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_f90_real(p, r, newtype, ierror)
    INTEGER, INTENT(IN) :: p, r
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_TYPE_CREATE_F90_REAL(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
```

This function returns a predefined MPI datatype that matches a REAL variable of KIND selected_real_kind(p, r). In the model described above it returns a handle for the element D(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. In communication, an MPI datatype A returned by MPI_TYPE_CREATE_F90_REAL matches a datatype B if and only if B was returned by 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 643.

It is erroneous to supply values for p and r not supported by the compiler.

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) |

**C binding**

```
int MPI_Type_create_f90_complex(int p, int r, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_f90_complex(p, r, newtype, ierror)
    INTEGER, INTENT(IN) ::  p, r
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_CREATE_F90_COMPLEX(P, R, NEWTYPE, IERROR)
    INTEGER P, R, NEWTYPE, IERROR
```

This function returns a predefined MPI datatype that matches a COMPLEX variable of KIND selected_real_kind(p, r). Either p or r may be omitted from calls to selected_real_kind(p, r) (but not both). Analogously, either p or r may be set to MPI_UNDEFINED. Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 643.

It is erroneous to supply values for p and r not supported by the compiler.

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) |

**C binding**

```
int MPI_Type_create_f90_integer(int r, MPI_Datatype *newtype)
```

**F08 binding**

```
MPI_Type_create_f90_integer(r, newtype, ierror)
    INTEGER, INTENT(IN) ::  r
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_CREATE_F90_INTEGER(R, NEWTYPE, IERROR)
    INTEGER R, NEWTYPE, IERROR
```

This function returns a predefined MPI datatype that matches a INTEGER variable of KIND selected_int_kind(r). Matching rules for datatypes created by this function are analogous to the matching rules for datatypes created by MPI_TYPE_CREATE_F90_REAL. Restrictions on using the returned datatype with the "external32" data representation are given on page 643.

It is erroneous to supply a value for r that is not supported by the compiler.

Example:

```
integer       longtype, quadtype
integer, parameter :: long = selected_int_kind(15)
integer(long) ii(10)
real(selected_real_kind(30)) x(10)
call MPI_TYPE_CREATE_F90_INTEGER(15, longtype, ierror)
call MPI_TYPE_CREATE_F90_REAL(30, MPI_UNDEFINED, quadtype, ierror)
...

call MPI_SEND(ii, 10, longtype, ...)
call MPI_SEND(x,  10, quadtype, ...)
```

*Advice to users.* The datatypes returned by the above functions are predefined datatypes. They cannot be freed; they do not need to be committed; they can be used with predefined reduction operations. There are two situations in which they behave differently syntactically, but not semantically, from the MPI named predefined datatypes.

1. MPI_TYPE_GET_ENVELOPE returns special combiners that allow a program to retrieve the values of p and r.
2. Because the datatypes are not named, they cannot be used as compile-time initializers or otherwise accessed before a call to one of the MPI_TYPE_CREATE_F90_XXX routines.

If a variable was declared specifying a non-default KIND value that was not obtained with selected_real_kind() or selected_int_kind(), the only way to obtain a matching MPI datatype is to use the size-based mechanism described in the next section.

(*End of advice to users.*)

*Advice to implementors.* An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, a high quality MPI implementation should return the same datatype handle for the same (REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination (p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

*Rationale.* The MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER interface needs as input the original range and precision values to be able to define useful and compiler-independent external (Section 13.5.2) or user-defined (Section 13.5.3) data representations, and in order to be able to perform automatic and efficient data conversions in a heterogeneous environment. (*End of rationale.*)

We now specify how the datatypes described in this section behave when used with the "external32" external data representation described in Section 13.5.2.

The external32 representation specifies data formats for integer and floating point values. Integer values are represented in two's complement big-endian format. Floating point values are represented by one of three IEEE formats. These are the IEEE "Single," "Double," and "Double Extended" formats, requiring 4, 8, and 16 bytes of storage, respectively. For the IEEE "Double Extended" formats, MPI specifies a Format Width of 16 bytes, with 15 exponent bits, bias = +10383, 112 fraction bits, and an encoding analogous to the "Double" format.

The external32 representations of the datatypes returned by MPI_TYPE_CREATE_F90_REAL/COMPLEX/INTEGER are given by the following rules.

For MPI_TYPE_CREATE_F90_REAL:

```
if      (p > 33) or (r > 4931) then  external32 representation
                                     is undefined
else if (p > 15) or (r >  307) then  external32_size = 16
else if (p >  6) or (r >   37) then  external32_size =  8
else                                 external32_size =  4
```

For MPI_TYPE_CREATE_F90_COMPLEX: twice the size as for MPI_TYPE_CREATE_F90_REAL.
For MPI_TYPE_CREATE_F90_INTEGER:

```
if      (r > 38) then  external32 representation is undefined
else if (r > 18) then  external32_size =  16
else if (r >  9) then  external32_size =  8
else if (r >  4) then  external32_size =  4
else if (r >  2) then  external32_size =  2
else                   external32_size =  1
```

If the external32 representation of a datatype is undefined, the result of using the datatype directly or indirectly (i.e., as part of another datatype or through a duplicated datatype) in operations that require the external32 representation is undefined. These operations include MPI_PACK_EXTERNAL, MPI_UNPACK_EXTERNAL, and many MPI_FILE functions, when the "external32" data representation is used. The ranges for which the external32 representation is undefined are reserved for future standardization.

### Support for Size-specific MPI Datatypes

MPI provides named datatypes corresponding to optional Fortran 77 numeric types that contain explicit byte lengths — MPI_REAL4, MPI_INTEGER8, etc. This section describes a mechanism that generalizes this model to support all Fortran numeric intrinsic types.

We assume that for each **typeclass** (integer, real, complex) and each word size there is a unique machine representation. For every pair (**typeclass**, **n**) supported by a compiler, MPI must provide a named size-specific datatype. The name of this datatype is of the form MPI_<TYPE>n in C and Fortran 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**) in Fortran. The list of names for such types includes:

```
MPI_REAL4
MPI_REAL8
MPI_REAL16
MPI_COMPLEX8
MPI_COMPLEX16
MPI_COMPLEX32
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_INTEGER8
MPI_INTEGER16
```

One datatype is required for each representation supported by the Fortran compiler.

> *Rationale.* Particularly for the longer floating-point types, C and Fortran may use different representations. For example, a Fortran compiler may define a 16-byte REAL type with 33 decimal digits of precision while a C compiler may define a 16-byte long double type that implements an 80-bit (10 byte) extended precision floating point value. Both of these types are 16 bytes long, but they are not interoperable. Thus, these types are defined by Fortran, even though C may define types of the same length. (*End of rationale.*)

To be backward compatible with the interpretation of these types in MPI-1, we assume that the nonstandard declarations `REAL*n`, `INTEGER*n`, always create a variable whose representation is of size **n**. These datatypes may also be used for variables declared with `KIND=INT8/16/32/64` or `KIND=REAL32/64/128`, which are defined in the `ISO_FORTRAN_ENV` intrinsic module. Note that the MPI datatypes and the `REAL*n`, `INTEGER*n` declarations count bytes whereas the Fortran `KIND` values count bits. All these datatypes are predefined.

The following 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) |

**F08 binding**

```
MPI_Sizeof(x, size, ierror)
    TYPE(*), DIMENSION(..)  ::  x
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
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 *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, datatype)

| | | |
|---|---|---|
| IN | typeclass | generic type specifier (integer) |
| IN | size | size, in bytes, of representation (integer) |
| OUT | datatype | datatype with correct type, size (handle) |

**C binding**

```
int MPI_Type_match_size(int typeclass, int size, MPI_Datatype *datatype)
```

**F08 binding**

```
MPI_Type_match_size(typeclass, size, datatype, ierror)
    INTEGER, INTENT(IN) ::  typeclass, size
    TYPE(MPI_Datatype), INTENT(OUT) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

**F binding**

```
MPI_TYPE_MATCH_SIZE(TYPECLASS, SIZE, DATATYPE, IERROR)
    INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

typeclass is one of MPI_TYPECLASS_REAL, MPI_TYPECLASS_INTEGER and MPI_TYPECLASS_COMPLEX, corresponding to the desired **typeclass**. The function returns an MPI datatype matching a local variable of type (**typeclass**, **size**).

This function returns a reference (handle) to one of the predefined named datatypes, not a duplicate. This type cannot be freed. MPI_TYPE_MATCH_SIZE can be used to obtain a size-specific type that matches a Fortran numeric intrinsic type by first calling MPI_SIZEOF in order to compute the variable size, and then calling MPI_TYPE_MATCH_SIZE to find a suitable datatype. In 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. ...
   }

   return MPI_SUCCESS;
}
```

(*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, communication 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 environment. Consider the following code:

```
real(selected_real_kind(5)) x(100)
call MPI_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)
```

```
if (myrank .eq. 0) then
    ... initialize x ...
    call MPI_SEND(x, xtype, 100, 1, ...)
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_SIZEOF(x, size, ierror)
call MPI_TYPE_MATCH_SIZE(MPI_TYPECLASS_REAL, size, xtype, ierror)

if (myrank .eq. 0) then
   call MPI_FILE_OPEN(MPI_COMM_SELF, 'foo',                &
                      MPI_MODE_CREATE+MPI_MODE_WRONLY,     &
                      MPI_INFO_NULL, fh, ierror)
   call MPI_FILE_SET_VIEW(fh, zero, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif

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, zero, xtype, xtype, 'external32',  &
                          MPI_INFO_NULL, ierror)
   call MPI_FILE_WRITE(fh, x, 100, xtype, status, ierror)
   call MPI_FILE_CLOSE(fh, ierror)
endif
```

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.*)

### 17.1.10 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 is intended to clarify, not add to, this standard.

As noted in the original MPI specification, the interface violates the Fortran standard in several ways. While these may cause few problems for Fortran 77 programs, they become more significant for Fortran 90 programs, so that users must exercise care when using new Fortran 90 features. With Fortran 2008 and the new semantics defined in TS 29113, most violations are resolved, and this is hinted at in an addendum to each item. The violations were originally adopted and have been retained because they are important for the usability of MPI. The rest of this section describes the potential problems in detail.

The following MPI features are inconsistent with Fortran 90 and Fortran 77.

1. An MPI subroutine with a choice argument may be called with different argument types. When using the `mpi_f08` module together with a compiler that supports Fortran 2008 + TS 29113, this problem is resolved.

2. An MPI subroutine with an assumed-size dummy argument may be passed an actual scalar argument. This is only solved for choice buffers through the use of `DIMENSION(..)`.

3. Nonblocking and split-collective MPI routines assume that actual arguments are passed by address or descriptor and that arguments and the associated data are not copied on entrance to or exit from the subroutine. This problem is solved with the use of the `ASYNCHRONOUS` attribute.

4. An MPI implementation may read or modify user data (e.g., communication buffers used by nonblocking communications) concurrently with a user program that is executing outside of MPI calls. This problem is resolved by relying on the extended semantics of the `ASYNCHRONOUS` attribute as specified in TS 29113.

5. Several named "constants," such as MPI_BOTTOM, MPI_IN_PLACE, MPI_STATUS_IGNORE, MPI_STATUSES_IGNORE, MPI_ERRCODES_IGNORE, MPI_UNWEIGHTED, MPI_WEIGHTS_EMPTY, MPI_ARGV_NULL, and MPI_ARGVS_NULL are not ordinary Fortran constants and require a special implementation. See Section 2.5.4 for more information.

6. The memory allocation routine MPI_ALLOC_MEM cannot be used from Fortran 77/90/95 without a language extension (for example, Cray pointers) that allows the allocated memory to be associated with a Fortran variable. Therefore, address sized integers were used in MPI-2.0 – MPI-2.2. In Fortran 2003, `TYPE(C_PTR)` entities were added, which allow a standard-conforming implementation of the semantics of MPI_ALLOC_MEM. In MPI-3.0 and later, MPI_ALLOC_MEM has an additional, overloaded interface to support this language feature. The use of Cray pointers is deprecated. The `mpi_f08` module only supports `TYPE(C_PTR)` pointers.

Additionally, MPI is inconsistent with Fortran 77 in a number of ways, as noted below.

- MPI identifiers exceed 6 characters.
- MPI identifiers may contain underscores after the first character.
- MPI requires an include file, mpif.h. On systems that do not support include files, the implementation should specify the values of named constants.
- Many routines in MPI have KIND-parameterized integers (e.g., MPI_ADDRESS_KIND and MPI_OFFSET_KIND) that hold address information. On systems that do not support Fortran 90-style parameterized types, INTEGER*8 or INTEGER should be used instead.

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 and Section 4.1.1 for more information.

Sections 17.1.11 through 17.1.19 describe several problems in detail which concern the interaction of MPI and Fortran as well as their solutions. Some of these solutions require special capabilities from the compilers. Major requirements are summarized in Section 17.1.7.

### 17.1.11 Problems Due to Strong Typing

All MPI functions with choice arguments associate actual arguments of different Fortran datatypes with the same dummy argument. This is not allowed by Fortran 77, and in Fortran 90, it is technically only allowed if the function is overloaded with a different function for each type (see also Section 17.1.6). In C, the use of void* formal arguments avoids these problems. Similar to C, with Fortran 2008 + TS 29113 (and later) together with the mpi_f08 module, the problem is avoided by declaring choice arguments with TYPE(*), DIMENSION(..), i.e., as assumed-type and assumed-rank dummy arguments.

Using INCLUDE 'mpif.h', the following code fragment is technically invalid and may generate a compile-time error.

```
integer i(5)
real    x(5)
...
call mpi_send(x, 5, MPI_REAL, ...)
call mpi_send(i, 5, MPI_INTEGER, ...)
```

In practice, it is rare for compilers to do more than issue a warning. When using either the mpi_f08 or mpi module, the problem is usually resolved through the assumed-type and assumed-rank declarations of the dummy arguments, or with a compiler-dependent mechanism that overrides type checking for choice arguments.

It is also technically invalid in Fortran to pass a scalar actual argument to an array dummy argument that is not a choice buffer argument. Thus, when using the mpi_f08 or mpi module, the following code fragment usually generates an error since the dims and periods arguments to MPI_CART_CREATE are declared as assumed size arrays INTEGER :: DIMS(*) and LOGICAL :: PERIODS(*).

```
USE mpi_f08     ! or  USE mpi
INTEGER size
CALL MPI_Cart_create(comm_old, 1, size, .TRUE., .TRUE., comm_cart, ierror)
```

Although this is a non-conforming MPI call, compiler warnings are not expected (but may occur) when using INCLUDE 'mpif.h' and this include file does not use Fortran explicit interfaces.

### 17.1.12 Problems Due to Data Copying and Sequence Association with Subscript Triplets

Arrays with subscript **triplets** describe Fortran subarrays with or without strides, e.g.,

```
REAL a(100,100,100)
CALL MPI_Send(a(11:17, 12:99:3, 1:100), 7*30*100, MPI_REAL, ...)
```

The handling of subscript triplets depends on the value of the constant MPI_SUBARRAYS_SUPPORTED:

- If MPI_SUBARRAYS_SUPPORTED equals .TRUE.:

  Choice buffer arguments are declared as TYPE(*), DIMENSION(..). For example, consider the following code fragment:

  ```
  REAL s(100), r(100)
  CALL MPI_Isend(s(1:100:5), 3, MPI_REAL, ..., rq, ierror)
  CALL MPI_Wait(rq, status, ierror)
  CALL MPI_Irecv(r(1:100:5), 3, MPI_REAL, ..., rq, ierror)
  CALL MPI_Wait(rq, status, ierror)
  ```

  In this case, the individual elements s(1), s(6), and s(11) are sent between the start of MPI_ISEND and the end of MPI_WAIT even though the compiled code will not copy s(1:100:5) to a real contiguous temporary scratch buffer. Instead, the compiled code will pass a descriptor to MPI_ISEND that allows MPI to operate directly on s(1), s(6), s(11), . . ., s(96). The called MPI_ISEND routine will take only the first three of these elements due to the type signature "3, MPI_REAL".

  All nonblocking MPI functions (e.g., MPI_ISEND, MPI_PUT, MPI_FILE_WRITE_ALL_BEGIN) behave as if *the user-specified elements of choice buffers are copied to a contiguous scratch buffer in the MPI runtime environment.* All datatype descriptions (in the example above, "3, MPI_REAL") read and store data from and to this virtual contiguous scratch buffer. Displacements in MPI derived datatypes are relative to the beginning of this virtual contiguous scratch buffer. Upon completion of a nonblocking receive operation (e.g., when MPI_WAIT on a corresponding MPI_Request returns), it is as if the received data has been copied from the virtual contiguous scratch buffer back to the non-contiguous application buffer. In the example above, r(1), r(6), and r(11) are guaranteed to be defined with the received data when MPI_WAIT returns.

  Note that the above definition does not supercede restrictions about buffers used with non-blocking operations (e.g., those specified in Section 3.7.2).

*Advice to implementors.* The Fortran descriptor for TYPE(*), DIMENSION(..) arguments contains enough information that, if desired, the MPI library can make a real contiguous copy of non-contiguous user buffers when the nonblocking operation is started, and release this buffer not before the nonblocking communication has completed (e.g., the MPI_WAIT routine). Efficient implementations may avoid such additional memory-to-memory data copying. (*End of advice to implementors.*)

*Rationale.* If MPI_SUBARRAYS_SUPPORTED equals .TRUE., non-contiguous buffers are handled inside the MPI library instead of by the compiler through argument association conventions. Therefore, the scope of MPI library scratch buffers can be from the beginning of a nonblocking operation until the completion of the operation although beginning and completion are implemented in different routines. (*End of rationale.*)

- If MPI_SUBARRAYS_SUPPORTED equals .FALSE.:

  In this case, the use of Fortran arrays with subscript triplets as actual choice buffer arguments in any nonblocking MPI operation (which also includes persistent request, and split collectives) may cause undefined behavior. They may, however, be used in blocking MPI operations.

  Implicit in MPI is the idea of a contiguous chunk of memory accessible through a linear address space. MPI copies data to and from this memory. An MPI program specifies the location of data by providing memory addresses and offsets. In the C language, sequence association rules plus pointers provide all the necessary low-level structure.

  In Fortran, array data is not necessarily stored contiguously. For example, the array section A(1:N:2) involves only the elements of A with indices 1, 3, 5, .... The same is true for a pointer array whose target is such a section. Most compilers ensure that an array that is a dummy argument is held in contiguous memory if it is declared with an explicit shape (e.g., B(N)) or is of assumed size (e.g., B(*)). If necessary, they do this by making a copy of the array into contiguous memory.[1]

  Because MPI dummy buffer arguments are assumed-size arrays if MPI_SUBARRAYS_SUPPORTED equals .FALSE., this leads to a serious problem for a nonblocking call: the compiler copies the temporary array back on return but MPI continues to copy data to the memory that held it. For example, consider the following code fragment:

```
real a(100)
call MPI_IRECV(a(1:100:2), MPI_REAL, 50, ...)
```

  Since the first dummy argument to MPI_IRECV is an assumed-size array (<type> buf(*)), the array section a(1:100:2) is copied to a temporary before being passed to MPI_IRECV, so that it is contiguous in memory. MPI_IRECV returns immediately, and data is copied from the temporary back into the array a. Sometime later, MPI may write to the address of the deallocated temporary. Copying is also a problem

[1] Technically, the Fortran standard is worded to allow non-contiguous storage of any array data, unless the dummy argument has the CONTIGUOUS attribute.

for MPI_ISEND since the temporary array may be deallocated before the data has all been sent from it.

Most Fortran 90 compilers do not make a copy if the actual argument is the whole of an explicit-shape or assumed-size array or is a “simply contiguous” section such as `A(1:N)` of such an array. (“Simply contiguous” is defined in the next paragraph.) Also, many compilers treat allocatable arrays the same as they treat explicit-shape arrays in this regard (though we know of one that does not). However, the same is not true for assumed-shape and pointer arrays; since they may be discontiguous, copying is often done. It is this copying that causes problems for MPI as described in the previous paragraph.

According to the Fortran 2008 Standard, Section 6.5.4, a “simply contiguous” array section is

```
name ( [:,]... [<subscript>]:[<subscript>] [,<subscript>]... )
```

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. The compiler can detect from analyzing the source code that the array is contiguous. 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 “simply contiguous” section of a contiguous array will also be contiguous.

The same problem can occur with a scalar argument. A compiler may make a copy of scalar dummy arguments within a called procedure when passed as an actual argument to a choice buffer routine. That this can cause a problem is illustrated by the example

```
real :: a
call user1(a,rq)
call MPI_WAIT(rq,status,ierr)
write (*,*) a

subroutine user1(buf,request)
call MPI_IRECV(buf,...,request,...)
end
```

If a is copied, MPI_IRECV will alter the copy when it completes the communication and will not alter a itself.

Note that copying will almost certainly occur for an argument that is a non-trivial expression (one with at least one operator or function call), a section that does not select a contiguous part of its parent (e.g., `A(1:n:2)`), a pointer whose target is such a section, or an assumed-shape array that is (directly or indirectly) associated with such a section.

If a compiler option exists that inhibits copying of arguments, in either the calling or called procedure, this must be employed.

If a compiler makes copies in the calling procedure of arguments that are explicit-shape or assumed-size arrays, "simply contiguous" array sections of such arrays, or scalars, and if no compiler option exists to inhibit such copying, then the compiler cannot be used for applications that use MPI_GET_ADDRESS, or any nonblocking MPI routine. If a compiler copies scalar arguments in the called procedure and there is no compiler option to inhibit this, then this compiler cannot be used for applications that use memory references across subroutine calls as in the example above.

### 17.1.13 Problems Due to Data Copying and Sequence Association with Vector Subscripts

Fortran arrays with **vector** subscripts describe subarrays containing a possibly irregular set of elements

```
REAL a(100)
CALL MPI_Send(A((/7,9,23,81,82/)), 5, MPI_REAL, ...)
```

Fortran arrays with a vector subscript must not be used as actual choice buffer arguments in any nonblocking or split collective MPI operations. They may, however, be used in blocking MPI operations.

### 17.1.14 Special Constants

MPI requires a number of special "constants" that cannot be implemented as normal Fortran constants, e.g., MPI_BOTTOM. The complete list can be found in Section 2.5.4. In C, these are implemented as constant pointers, usually as NULL and are used where the function prototype calls for a pointer to a variable, not the variable itself.

In Fortran, using special values for the constants (e.g., by defining them through parameter statements) is not possible because an implementation cannot distinguish these values from valid data. Typically these constants are implemented as predefined static variables (e.g., a variable in an MPI-declared COMMON block), relying on the fact that the target compiler passes data by address. Inside the subroutine, the address of the actual choice buffer argument can be compared with the address of such a predefined static variable.

These special constants also cause an exception with the usage of Fortran INTENT: with USE mpi_f08, the attributes INTENT(IN), INTENT(OUT), and INTENT(INOUT) are used in the Fortran interface. In most cases, INTENT(IN) is used if the C interface uses call-by-value. For all buffer arguments and for dummy arguments that may be modified and allow one of these special constants as input, an INTENT is not specified.

### 17.1.15 Fortran Derived Types

MPI supports passing Fortran entities of BIND(C) and SEQUENCE derived types to choice dummy arguments, provided no type component has the ALLOCATABLE or POINTER attribute.

The following code fragment shows some possible ways to send scalars or arrays of interoperable derived type in Fortran. The example assumes that all data is passed by address.

```
type, BIND(C) :: mytype
   integer :: i
   real :: x
```

```
   double precision :: d
   logical :: l
end type mytype

type(mytype) :: foo, fooarr(5)
integer :: blocklen(4), type(4)
integer(KIND=MPI_ADDRESS_KIND) :: disp(4), base, lb, extent

call MPI_GET_ADDRESS(foo%i, disp(1), ierr)
call MPI_GET_ADDRESS(foo%x, disp(2), ierr)
call MPI_GET_ADDRESS(foo%d, disp(3), ierr)
call MPI_GET_ADDRESS(foo%l, disp(4), ierr)

base = disp(1)
disp(1) = disp(1) - base
disp(2) = disp(2) - base
disp(3) = disp(3) - base
disp(4) = disp(4) - base

blocklen(1) = 1
blocklen(2) = 1
blocklen(3) = 1
blocklen(4) = 1

type(1) = MPI_INTEGER
type(2) = MPI_REAL
type(3) = MPI_DOUBLE_PRECISION
type(4) = MPI_LOGICAL

call MPI_TYPE_CREATE_STRUCT(4, blocklen, disp, type, newtype, ierr)
call MPI_TYPE_COMMIT(newtype, ierr)

call MPI_SEND(foo%i, 1, newtype, dest, tag, comm, ierr)
! or
call MPI_SEND(foo, 1, newtype, dest, tag, comm, ierr)
! expects that base == address(foo%i) == address(foo)

call MPI_GET_ADDRESS(fooarr(1), disp(1), ierr)
call MPI_GET_ADDRESS(fooarr(2), disp(2), ierr)
extent = disp(2) - disp(1)
lb = 0
call MPI_TYPE_CREATE_RESIZED(newtype, lb, extent, newarrtype, ierr)
call MPI_TYPE_COMMIT(newarrtype, ierr)

call MPI_SEND(fooarr, 5, newarrtype, dest, tag, comm, ierr)
```

Using the derived type variable `foo` instead of its first basic type element `foo%i` may be impossible if the MPI library implements choice buffer arguments through overloading instead of using TYPE(*), DIMENSION(..), or through a non-standardized extension such as !$PRAGMA IGNORE_TKR; see Section 17.1.6.

To use a derived type in an array requires a correct extent of the datatype handle to take care of the alignment rules applied by the compiler. These alignment rules may imply that there are gaps between the components of a derived type, and also between the subsuquent elements of an array of a derived type. The extent of an interoperable derived type (i.e., defined with BIND(C)) and a SEQUENCE derived type with the same content may be different because C and Fortran may apply different alignment rules. As recommended in the advice to users in Section 4.1.6, one should add an additional fifth structure element with one numerical storage unit at the end of this structure to force in most cases that the array of structures is contiguous. Even with such an additional element, one should keep this resizing due to the special alignment rules that can be used by the compiler for structures, as also mentioned in this advice.

Using the extended semantics defined in TS 29113, it is also possible to use entities or derived types without either the BIND(C) or the SEQUENCE attribute as choice buffer arguments; some additional constraints must be observed, e.g., no ALLOCATABLE or POINTER type components may exist. In this case, the base address in the example must be changed to become the address of foo instead of foo%i, because the Fortran compiler may rearrange type components or add padding. Sending the structure foo should then also be performed by providing it (and not foo%i) as actual argument for MPI_Send.

### 17.1.16 Optimization Problems, an Overview

MPI provides operations that may be hidden from the user code and run concurrently with it, accessing the same memory as user code. Examples include the data transfer for an MPI_IRECV. The optimizer of a compiler will assume that it can recognize periods when a copy of a variable can be kept in a register without reloading from or storing to memory. When the user code is working with a register copy of some variable while the hidden operation reads or writes the memory copy, problems occur. These problems are independent of the Fortran support method; i.e., they occur with the mpi_f08 module, the mpi module, and the mpif.h include file.

This section shows four problematic usage areas (the abbrevations in parentheses are used in the table below):

- Use of nonblocking routines or persistent requests *(Nonbl.)*.
- Use of one-sided routines *(1-sided)*.
- Use of MPI parallel file I/O split collective operations *(Split)*.
- Use of MPI_BOTTOM together with absolute displacements in MPI datatypes, or relative displacements between two variables in such datatypes *(Bottom)*.

The following compiler optimization strategies (valid for serial code) may cause problems in MPI applications:

- Code movement and register optimization problems; see Section 17.1.17.
- Temporary data movement and temporary memory modifications; see Section 17.1.18.
- Permanent data movement (e.g., through garbage collection); see Section 17.1.19.

| Optimization ... | ... may cause a problem in following usage areas | | | |
|---|---|---|---|---|
| | Nonbl. | 1-sided | Split | Bottom |
| Code movement and register optimization | yes | yes | no | yes |
| Temporary data movement | yes | yes | yes | no |
| Permanent data movement | yes | yes | yes | yes |

Table 17.2: Occurrence of Fortran optimization problems in several usage areas

Table 17.2 shows the only usage areas where these optimization problems may occur.

The solutions in the following sections are based on compromises:

- to minimize the burden for the application programmer, e.g., as shown in Sections "Solutions" through "The (Poorly Performing) Fortran VOLATILE Attribute" on pages 658–663,
- to minimize the drawbacks on compiler based optimization, and
- to minimize the requirements defined in Section 17.1.7.

### 17.1.17 Problems with Code Movement and Register Optimization

#### Nonblocking Operations

If a variable is local to a Fortran subroutine (i.e., not in a module or a `COMMON` block), the compiler will assume that it cannot be modified by a called subroutine unless it is an actual argument of the call. In the most common linkage convention, the subroutine is expected to save and restore certain registers. Thus, the optimizer will assume that a register which held a valid copy of such a variable before the call will still hold a valid copy on return.

**Example 17.1** Fortran 90 register optimization — extreme.

```
Source                     compiled as                or compiled as

REAL :: buf, b1            REAL :: buf, b1            REAL :: buf, b1
call MPI_IRECV(buf,..req)  call MPI_IRECV(buf,..req)  call MPI_IRECV(buf,..req)
                           register = buf             b1 = buf
call MPI_WAIT(req,..)      call MPI_WAIT(req,..)      call MPI_WAIT(req,..)
b1 = buf                   b1 = register
```

Example 17.1 shows extreme, but allowed, possibilities. MPI_WAIT on a concurrent thread modifies buf between the invocation of MPI_IRECV and the completion of MPI_WAIT. But the compiler cannot see any possibility that buf can be changed after MPI_IRECV has returned, and may schedule the load of buf earlier than typed in the source. The compiler has no reason to avoid using a register to hold buf across the call to MPI_WAIT. It also may reorder the instructions as illustrated in the rightmost column.

Due to valid compiler code movement optimizations in Example 17.2, the content of buf may already have been overwritten by the compiler when the content of buf is sent.

**Example 17.2** Similar example with MPI_ISEND

| Source | compiled as | with a possible MPI-internal execution sequence |
|---|---|---|

```
REAL :: buf, copy            REAL :: buf, copy            REAL :: buf, copy
buf = val                    buf = val                    buf = val
call MPI_ISEND(buf,..req)    call MPI_ISEND(buf,..req)    addr = &buf
copy = buf                   copy= buf                    copy = buf
                             buf = val_overwrite          buf = val_overwrite
call MPI_WAIT(req,..)        call MPI_WAIT(req,..)        call send(*addr) ! within
                                                                           ! MPI_WAIT
buf = val_overwrite
```

The code movement is permitted because the compiler cannot detect a possible access to buf in MPI_WAIT (or in a second thread between the start of MPI_ISEND and the end of MPI_WAIT).

Such register optimization is based on moving code; here, the access to buf was moved from after MPI_WAIT to before MPI_WAIT. Note that code movement may also occur across subroutine boundaries when subroutines or functions are inlined.

This register optimization/code movement problem for nonblocking operations does not occur with MPI parallel file I/O split collective operations, because in the ..._BEGIN and ..._END calls, the same buffer has to be provided as an actual argument. The register optimization / code movement problem for MPI_BOTTOM and derived MPI datatypes may occur in each blocking and nonblocking communication call, as well as in each parallel file I/O operation.

#### Persistent Operations

With persistent requests, the buffer argument is hidden from the MPI_START and MPI_STARTALL calls, i.e., the Fortran compiler may move buffer accesses across the MPI_START or MPI_STARTALL call, similar to the MPI_WAIT call as described in the Nonblocking Operations subsection in Section 17.1.17.

#### One-sided Communication

An example with instruction reordering due to register optimization can be found in Section 11.7.4.

#### MPI_BOTTOM and Combining Independent Variables in Datatypes

This section is only relevant if the MPI program uses a buffer argument to an MPI_SEND, MPI_RECV, etc., that hides the actual variables involved in the communication. MPI_BOTTOM with an MPI_Datatype containing *absolute addresses* is one example. Creating a datatype which uses one variable as an anchor and brings along others by using MPI_GET_ADDRESS to determine their offsets from the anchor is another. The anchor variable would be the only one referenced in the call. Also attention must be paid if MPI operations are used that run in parallel with the user's application.

Example 17.3 shows what Fortran compilers are allowed to do.

**Example 17.3** Fortran 90 register optimization.

```
This source ...                          can be compiled as:

call MPI_GET_ADDRESS(buf,bufaddr,        call MPI_GET_ADDRESS(buf,...)
               ierror)
call MPI_TYPE_CREATE_STRUCT(1,1,         call MPI_TYPE_CREATE_STRUCT(...)
               bufaddr,
               MPI_REAL,type,ierror)
call MPI_TYPE_COMMIT(type,ierror)        call MPI_TYPE_COMMIT(...)
val_old = buf                            register = buf
                                         val_old = register
call MPI_RECV(MPI_BOTTOM,1,type,...)     call MPI_RECV(MPI_BOTTOM,...)
val_new = buf                            val_new = register
```

In Example 17.3, the compiler does not invalidate the register because it cannot see that MPI_RECV changes the value of buf. The access to buf is hidden by the use of MPI_GET_ADDRESS and MPI_BOTTOM.

**Example 17.4** Similar example with MPI_SEND

```
This source ...                          can be compiled as:

! buf contains val_old                   ! buf contains val_old
buf = val_new
call MPI_SEND(MPI_BOTTOM,1,type,...)     call MPI_SEND(...)
! with buf as a displacement in type     ! i.e. val_old is sent
                                         !
                                         ! buf=val_new  is moved to here
                                         ! and detected as dead code
                                         ! and therefore removed
                                         !
buf = val_overwrite                      buf = val_overwrite
```

In Example 17.4, several successive assignments to the same variable buf can be combined in a way such that only the last assignment is executed. "Successive" means that no interfering load access to this variable occurs between the assignments. The compiler cannot detect that the call to MPI_SEND statement is interfering because the load access to buf is hidden by the usage of MPI_BOTTOM.

### Solutions

The following sections show in detail how the problems with code movement and register optimization can be portably solved. Application writers can partially or fully avoid these compiler optimization problems by using one or more of the special Fortran declarations with the send and receive buffers used in nonblocking operations, or in operations in which MPI_BOTTOM is used, or if datatype handles that combine several variables are used:

- Use of the Fortran `ASYNCHRONOUS` attribute.
- Use of the helper routine MPI_F_SYNC_REG, or an equivalent user-written dummy routine.
- Declare the buffer as a Fortran module variable or within a Fortran common block.
- Use of the Fortran `VOLATILE` attribute.

**Example 17.5** Protecting nonblocking communication with the `ASYNCHRONOUS` attribute.

```
USE mpi_f08
REAL, ASYNCHRONOUS :: b(0:101) ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)            ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left,right,...)
CALL MPI_Irecv(b(  0), ..., left,  ..., req(1), ...)
CALL MPI_Irecv(b(101), ..., right, ..., req(2), ...)
CALL MPI_Isend(b(  1), ..., left,  ..., req(3), ...)
CALL MPI_Isend(b(100), ..., right, ..., req(4), ...)

#ifdef WITHOUT_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (a)
  CALL MPI_Waitall(4, req, ...)
  DO i=1,100 ! compute all new local data
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
#endif

#ifdef WITH_OVERLAPPING_COMMUNICATION_AND_COMPUTATION
! Case (b)
  DO i=2,99  ! compute  only elements for which halo data is not needed
    bnew(i) = function(b(i-1), b(i), b(i+1))
  END DO
  CALL MPI_Waitall(4, req, ...)
  i=1 ! compute leftmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
  i=100 ! compute rightmost element
    bnew(i) = function(b(i-1), b(i), b(i+1))
#endif
```

Each of these methods solves the problems of code movement and register optimization, but may incur various degrees of performance impact, and may not be usable in every application context. These methods may not be guaranteed by the Fortran standard, but they must be guaranteed by a MPI-3.0 (and later) compliant MPI library and associated compiler suite according to the requirements listed in Section 17.1.7. The performance impact of using MPI_F_SYNC_REG is expected to be low, that of using module variables or the ASYNCHRONOUS attribute is expected to be low to medium, and that of using the VOLATILE attribute is expected to be high or very high. Note that there is one attribute that cannot be used for this purpose: the Fortran TARGET attribute does not solve code movement problems in MPI applications.

#### The Fortran ASYNCHRONOUS Attribute

Declaring an actual buffer argument with the ASYNCHRONOUS Fortran attribute in a scoping unit (or BLOCK) informs the compiler that any statement in the scoping unit may be executed while the buffer is affected by a pending asynchronous Fortran input/output operation (since Fortran 2003) or by an asynchronous communication (TS 29113 extension). Without the extensions specified in TS 29113, a Fortran compiler may totally ignore this attribute if the Fortran compiler implements asynchronous Fortran input/output operations with blocking I/O. The ASYNCHRONOUS attribute protects the buffer accesses from optimizations through code movements across routine calls, and the buffer itself from temporary and permanent data movements. If the choice buffer dummy argument of a nonblocking MPI routine is declared with ASYNCHRONOUS (which is mandatory for the mpi_f08 module, with allowable exceptions listed in Section 17.1.6), then the compiler has to guarantee call by reference and should report a compile-time error if call by reference is impossible, e.g., if vector subscripts are used. The MPI_ASYNC_PROTECTS_NONBLOCKING is set to .TRUE. if both the protection of the actual buffer argument through ASYNCHRONOUS according to the TS 29113 extension and the declaration of the dummy argument with ASYNCHRONOUS in the Fortran support method is guaranteed for all nonblocking routines, otherwise it is set to .FALSE..

The ASYNCHRONOUS attribute has some restrictions. Section 5.4.2 of the TS 29113 specifies:

> "Asynchronous communication for a Fortran variable occurs through the action of procedures defined by means other than Fortran. It is initiated by execution of an asynchronous communication initiation procedure and completed by execution of an asynchronous communication completion procedure. Between the execution of the initiation and completion procedures, any variable of which any part is associated with any part of the asynchronous communication variable is a pending communication affector. Whether a procedure is an asynchronous communication initiation or completion procedure is processor dependent.
>
> Asynchronous communication is either input communication or output communication. For input communication, a pending communication affector shall not be referenced, become defined, become undefined, become associated with a dummy argument that has the VALUE attribute, or have its pointer association status changed. For output communication, a pending communication affector shall not be redefined, become undefined, or have its pointer association status changed."

In Example 17.5 Case (a) on page 659, the read accesses to b within function(b(i-1), b(i), b(i+1)) cannot be moved by compiler optimizations to before the wait call because b was declared as ASYNCHRONOUS. Note that only the elements 0, 1, 100, and 101 of b are involved in asynchronous communication but by definition, the total variable b is the pending communication affector and is usable for input and output asynchronous communication between the MPI_I... routines and MPI_Waitall. Case (a) works fine because the read accesses to b occur after the communication has completed.

In Case (b), the read accesses to b(1:100) in the loop i=2,99 are read accesses to a pending communication affector while input communication (i.e., the two MPI_Irecv calls) is pending. This is a contradiction to the rule that *for input communication, a pending communication affector shall not be referenced*. The problem can be solved by using separate variables for the halos and the inner array, or by splitting a common array into disjoint subarrays which are passed through different dummy arguments into a subroutine, as shown in Example 17.9.

If one does not overlap communication and computation on the same variable, then all optimization problems can be solved through the ASYNCHRONOUS attribute.

The problems with MPI_BOTTOM, as shown in Example 17.3 and Example 17.4, can also be solved by declaring the buffer buf with the ASYNCHRONOUS attribute.

In some MPI routines, a buffer dummy argument is defined as ASYNCHRONOUS to guarantee passing by reference, provided that the actual argument is also defined as ASYNCHRONOUS.

#### Calling MPI_F_SYNC_REG

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. The MPI library provides the MPI_F_SYNC_REG routine for this purpose; see Section 17.1.8.

- The problems illustrated by the Examples 17.1 and 17.2 can be solved by calling MPI_F_SYNC_REG(buf) once immediately after MPI_WAIT.

  Example 17.1
  can be solved with

  ```
  call MPI_IRECV(buf,..req)


  call MPI_WAIT(req,..)
  call MPI_F_SYNC_REG(buf)
  b1 = buf
  ```

  Example 17.2
  can be solved with

  ```
  buf = val
  call MPI_ISEND(buf,..req)
  copy = buf
  call MPI_WAIT(req,..)
  call MPI_F_SYNC_REG(buf)
  buf = val_overwrite
  ```

  The call to MPI_F_SYNC_REG(buf) prevents moving the last line before the MPI_WAIT call. Further calls to MPI_F_SYNC_REG(buf) are not needed because it is still correct if the additional read access copy=buf is moved below MPI_WAIT and before buf=val_overwrite.

- The problems illustrated by the Examples 17.3 and 17.4 can be solved with two additional MPI_F_SYNC_REG(buf) statements; one directly before MPI_RECV/ MPI_SEND, and one directly after this communication operation.

  Example 17.3
  can be solved with

  ```
  call MPI_F_SYNC_REG(buf)
  call MPI_RECV(MPI_BOTTOM,...)
  call MPI_F_SYNC_REG(buf)
  ```

  Example 17.4
  can be solved with

  ```
  call MPI_F_SYNC_REG(buf)
  call MPI_SEND(MPI_BOTTOM,...)
  call MPI_F_SYNC_REG(buf)
  ```

The first call to MPI_F_SYNC_REG(buf) is needed to finish all load and store references to buf prior to MPI_RECV/MPI_SEND; the second call is needed to assure that any subsequent access to buf is not moved before MPI_RECV/SEND.

- In the example in Section 11.7.4, two asynchronous accesses must be protected: in Process 1, the access to bbbb must be protected similar to Example 17.1, i.e., a call to MPI_F_SYNC_REG(bbbb) is needed after the second MPI_WIN_FENCE to guarantee that further accesses to bbbb are not moved ahead of the call to MPI_WIN_FENCE. In Process 2, both calls to MPI_WIN_FENCE together act as a communication call with MPI_BOTTOM as the buffer. That is, before the first fence and after the second fence, a call to MPI_F_SYNC_REG(buff) is needed to guarantee that accesses to buff are not moved after or ahead of the calls to MPI_WIN_FENCE. Using MPI_GET instead of MPI_PUT, the same calls to MPI_F_SYNC_REG are necessary.

| **Source of Process 1** | **Source of Process 2** |
|---|---|
| `bbbb = 777` | `buff = 999` |
| | `call MPI_F_SYNC_REG(buff)` |
| `call MPI_WIN_FENCE` | `call MPI_WIN_FENCE` |
| `call MPI_PUT(bbbb` | |
| `into buff of process 2)` | |
| | |
| `call MPI_WIN_FENCE` | `call MPI_WIN_FENCE` |
| `call MPI_F_SYNC_REG(bbbb)` | `call MPI_F_SYNC_REG(buff)` |
| | `ccc = buff` |

- The temporary memory modification problem, i.e., Example 17.6, can **not** be solved with this method.

#### A User Defined Routine Instead of MPI_F_SYNC_REG

Instead of MPI_F_SYNC_REG, one can also use a user defined external subroutine, which is separately compiled:

```
subroutine DD(buf)
  integer buf
end
```

Note that if the intent is declared in an explicit interface for the external subroutine, it must be `OUT` or `INOUT`. The subroutine itself may have an empty body, but the compiler does not know this and has to assume that the buffer may be altered. For example, a call to MPI_RECV with MPI_BOTTOM as buffer might be replaced by

```
call DD(buf)
call MPI_RECV(MPI_BOTTOM,...)
call DD(buf)
```

Such a user-defined routine was introduced in MPI-2.0 and is still included here to document such usage in existing application programs although new applications should prefer MPI_F_SYNC_REG or one of the other possibilities. In an existing application, calls to such a user-written routine should be substituted by a call to MPI_F_SYNC_REG because the user-written routine may not be implemented in accordance with the rules specified in Section 17.1.7.

#### Module Variables and COMMON Blocks

An alternative to the previously mentioned methods 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 may alter the buffer or variable, provided that the compiler cannot infer that the MPI procedure does not reference the module or common block.

- This method solves problems of instruction reordering, code movement, and register optimization related to nonblocking and one-sided communication, or related to the usage of MPI_BOTTOM and derived datatype handles.
- Unfortunately, this method does **not** solve problems caused by asynchronous accesses between the start and end of a nonblocking or one-sided communication. Specifically, problems caused by temporary memory modifications are not solved.

#### The (Poorly Performing) Fortran VOLATILE Attribute

The `VOLATILE` attribute gives the buffer or variable the properties needed to avoid register optimization or code movement problems, but it may inhibit optimization of any code containing references or definitions of the buffer or variable. On many modern systems, the performance impact will be large because not only register, but also cache optimizations will not be applied. Therefore, use of the `VOLATILE` attribute to enforce correct execution of MPI programs is discouraged.

#### The Fortran TARGET Attribute

The `TARGET` attribute does not solve the code movement problem because it is not specified for the choice buffer dummy arguments of nonblocking routines. If the compiler detects that the application program specifies the `TARGET` attribute for an actual buffer argument used in the call to a nonblocking routine, the compiler may ignore this attribute if no pointer reference to this buffer exists.

> *Rationale.* The Fortran standardization body decided to extend the `ASYNCHRONOUS` attribute within the TS 29113 to protect buffers in nonblocking calls from all kinds of optimization, instead of extending the `TARGET` attribute. (*End of rationale.*)

### 17.1.18 Temporary Data Movement and Temporary Memory Modification

The compiler is allowed to temporarily modify data in memory. Normally, this problem may occur only when overlapping communication and computation, as in Example 17.5, Case (b) on page 659. Example 17.6 also shows a possibility that could be problematic.

In the compiler-generated, possible optimization in Example 17.7, buf(100,100) from Example 17.6 is equivalenced with the 1-dimensional array buf_1dim(10000). The nonblocking receive may asynchronously receive the data in the boundary buf(1,1:100) while the fused

**Example 17.6** Overlapping Communication and Computation.

```
USE mpi_f08
REAL :: buf(100,100)
CALL MPI_Irecv(buf(1,1:100),..., req,...)
DO j=1,100
  DO i=2,100
    buf(i,j)=...
  END DO
END DO
CALL MPI_Wait(req,...)
```

**Example 17.7** The compiler may substitute the nested loops through loop fusion.

```
REAL :: buf(100,100),  buf_1dim(10000)
EQUIVALENCE (buf(1,1), buf_1dim(1))
CALL MPI_Irecv(buf(1,1:100),..., req,...)
tmp(1:100) = buf(1,1:100)
DO j=1,10000
  buf_1dim(h)=...
END DO
buf(1,1:100) = tmp(1:100)
CALL MPI_Wait(req,...)
```

loop is temporarily using this part of the buffer. When the tmp data is written back to buf, the previous data of buf(1,1:100) is restored and the received data is lost. The principle behind this optimization is that the receive buffer data buf(1,1:100) was temporarily moved to tmp.

Example 17.8 shows a second possible optimization. The whole array is temporarily moved to local_buf.

When storing local_buf back to the original location buf, then this implies overwriting the section of buf that serves as a receive buffer in the nonblocking MPI call, i.e., this storing back of local_buf is therefore likely to interfere with asynchronously received data in buf(1,1:100).

Note that this problem may also occur:

- With the local buffer at the origin process, between an RMA communication call and the ensuing synchronization call; see Chapter 11.
- With the window buffer at the target process between two ensuing RMA synchronization calls.
- With the local buffer in MPI parallel file I/O split collective operations between the ..._BEGIN and ..._END calls; see Section 13.4.5.

As already mentioned in subsection *The Fortran ASYNCHRONOUS attribute* on page 660 of Section 17.1.17, the ASYNCHRONOUS attribute can prevent compiler optimization

**Example 17.8** Another optimization is based on the usage of a separate memory storage area, e.g., in a GPU.

```
REAL :: buf(100,100), local_buf(100,100)
CALL MPI_Irecv(buf(1,1:100),..., req,...)
local_buf = buf
DO j=1,100
  DO i=2,100
    local_buf(i,j)=...
  END DO
END DO
buf = local_buf ! may overwrite asynchronously received
                ! data in buf(1,1:100)
CALL MPI_Wait(req,...)
```

with temporary data movement, but only if the receive buffer and the local references are separated into different variables, as shown in Example 17.9 and in Example 17.10.

Note also that the methods

- calling MPI_F_SYNC_REG (or such a user-defined routine),
- using module variables and `COMMON` blocks, and
- the `TARGET` attribute

cannot be used to prevent such temporary data movement. These methods influence compiler optimization when library routines are called. They cannot prevent the optimizations of the code fragments shown in Example 17.6 and 17.7.

Note also that compiler optimization with temporary data movement should **not** be prevented by declaring `buf` as `VOLATILE` because the `VOLATILE` implies that all accesses to any storage unit (word) of `buf` must be directly done in the main memory exactly in the sequence defined by the application program. The `VOLATILE` attribute prevents all register and cache optimizations. Therefore, `VOLATILE` may cause a huge performance degradation.

Instead of solving the problem, it is better to **prevent** the problem: when overlapping communication and computation, the nonblocking communication (or nonblocking or split collective I/O) and the computation should be executed **on different variables**, and the communication should be *protected* with the `ASYNCHRONOUS` attribute. In this case, the temporary memory modifications are done only on the variables used in the computation and cannot have any side effect on the data used in the nonblocking MPI operations.

*Rationale.* This is a strong restriction for application programs. To weaken this restriction, a new or modified asynchronous feature in the Fortran language would be necessary: an asynchronous attribute that can be used on parts of an array and together with asynchronous operations outside the scope of Fortran. If such a feature becomes available in a future edition of the Fortran standard, then this restriction also may be weakened in a later version of the MPI standard. (*End of rationale.*)

In Example 17.9 (which is a solution for the problem shown in Example 17.5 and in Example 17.10 (which is a solution for the problem shown in Example 17.8), the array is split into inner and halo part and both disjoint parts are passed to a subroutine separated_sections. This routine overlaps the receiving of the halo data and the calculations on the inner part of the array. In a second step, the whole array is used to do the calculation on the elements where inner+halo is needed. Note that the halo and the inner area are strided arrays. Those can be used in non-blocking communication only with a TS 29113 based MPI library.

### 17.1.19 Permanent Data Movement

A Fortran compiler may implement permanent data movement during the execution of a Fortran program. This would require that pointers to such data are appropriately updated. An implementation with automatic garbage collection is one use case. Such permanent data movement is in conflict with MPI in several areas:

- MPI datatype handles with absolute addresses in combination with MPI_BOTTOM.
- All nonblocking MPI operations if the internally used pointers to the buffers are not updated by the Fortran runtime, or if within an MPI process, the data movement is executed in parallel with the MPI operation.

This problem can be also solved by using the ASYNCHRONOUS attribute for such buffers. This MPI standard requires that the problems with permanent data movement do not occur by imposing suitable restrictions on the MPI library together with the compiler used; see Section 17.1.7.

### 17.1.20 Comparison with C

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 indirection on 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. Problems due to temporary memory modifications can also occur in C. As above, the best advice is to avoid the problem: use different variables for buffers in nonblocking MPI operations and computation that is executed while a nonblocking operation is pending.

**Example 17.9** Using separated variables for overlapping communication and computation to allow the protection of nonblocking communication with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL :: b(0:101)     ! elements 0 and 101 are halo cells
REAL :: bnew(0:101)  ! elements 1 and 100 are newly computed
INTEGER :: i
CALL separated_sections(b(0), b(1:100), b(101), bnew(0:101))
i=1 ! compute leftmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
i=100 ! compute rightmost element
  bnew(i) = function(b(i-1), b(i), b(i+1))
END

SUBROUTINE separated_sections(b_lefthalo, b_inner, b_righthalo, bnew)
USE mpi_f08
REAL, ASYNCHRONOUS :: b_lefthalo(0:0), b_inner(1:100), b_righthalo(101:101)
REAL :: bnew(0:101)  ! elements 1 and 100 are newly computed
TYPE(MPI_Request) :: req(4)
INTEGER :: left, right, i
CALL MPI_Cart_shift(...,left, right,...)
CALL MPI_Irecv(b_lefthalo (  0), ..., left,  ..., req(1), ...)
CALL MPI_Irecv(b_righthalo(101), ..., right, ..., req(2), ...)
! b_lefthalo and b_righthalo is written asynchronously.
! There is no other concurrent access to b_lefthalo and b_righthalo.
CALL MPI_Isend(b_inner(  1),     ..., left,  ..., req(3), ...)
CALL MPI_Isend(b_inner(100),     ..., right, ..., req(4), ...)

DO i=2,99  ! compute  only elements for which halo data is not needed
  bnew(i) = function(b_inner(i-1), b_inner(i), b_inner(i+1))
  ! b_inner is read and sent at the same time.
  ! This is allowed based on the rules for ASYNCHRONOUS.
END DO
CALL MPI_Waitall(4, req,...)
END SUBROUTINE
```

**Example 17.10** Protecting GPU optimizations with the ASYNCHRONOUS attribute.

```
USE mpi_f08
REAL :: buf(100,100)
CALL separated_sections(buf(1:1,1:100), buf(2:100,1:100))
END

SUBROUTINE separated_sections(buf_halo, buf_inner)
REAL, ASYNCHRONOUS :: buf_halo(1:1,1:100)
REAL :: buf_inner(2:100,1:100)
REAL :: local_buf(2:100,100)

CALL MPI_Irecv(buf_halo(1,1:100),..., req,...)
local_buf = buf_inner
DO j=1,100
  DO i=2,100
    local_buf(i,j)=...
  END DO
END DO
buf_inner = local_buf ! buf_halo is not touched!!!

CALL MPI_Wait(req,...)
```

## 17.2 Language Interoperability

### 17.2.1 Introduction

It is not uncommon for library developers to use one language to develop an application library that may be called by an application program written in a different language. MPI currently supports ISO (previously ANSI) C and Fortran bindings. It should be possible for applications in any of the supported languages to call MPI-related functions in another language.

Moreover, MPI allows the development of client-server code, with MPI communication used between a parallel client and a parallel server. It should be possible to code the server in one language and the clients in another language. To do so, communications should be possible between applications written in different languages.

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 extensible to new languages, should MPI bindings be defined for such languages.

### 17.2.2 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 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 `INTEGER`s, can be passed to a C program. We also assume that Fortran 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.

### 17.2.3 Initialization

A call to MPI_INIT or MPI_INIT_THREAD, from any language, initializes MPI for execution in all languages.

*Advice to users.* Certain implementations use the (inout) `argc`, `argv` arguments of the C version of MPI_INIT in order to propagate values for `argc` and `argv` to all executing processes. Use of the Fortran version of MPI_INIT to initialize MPI may result in a loss of this ability. (*End of advice to users.*)

The function MPI_INITIALIZED returns the same answer in all languages.

The function MPI_FINALIZE finalizes the MPI environments for all languages.

The function MPI_FINALIZED returns the same answer in all languages.

The function MPI_ABORT kills processes, irrespective of the language used by the caller or by the processes killed.

The MPI environment is initialized in the same manner for all languages by MPI_INIT. E.g., MPI_COMM_WORLD carries the same information regardless of language: same processes, same environmental attributes, same error handlers.

Information can be added to info objects in one language and retrieved in another.

*Advice to users.* The use of several languages in one MPI program may require the use of special options at compile and/or link time. (*End of advice to users.*)

*Advice to implementors.* Implementations may selectively link language specific MPI libraries only to codes that need them, so as not to increase the size of binaries for codes that use only one language. The MPI initialization code need perform initialization for a language only if that language library is loaded. (*End of advice to implementors.*)

### 17.2.4 Transfer of Handles

Handles are passed between Fortran and C by using an explicit C wrapper to convert Fortran handles to C handles. There is no direct access to C handles in Fortran.

The type definition MPI_Fint is provided in C for an integer of the size that matches a Fortran `INTEGER`; usually, MPI_Fint will be equivalent to `int`. With the Fortran `mpi` module or the `mpif.h` include file, a Fortran handle is a Fortran `INTEGER` value that can be used in the following conversion functions. With the Fortran `mpi_f08` module, a Fortran handle is a `BIND(C)` derived type that contains an `INTEGER` component named MPI_VAL. This `INTEGER` value can be used in the following conversion functions.

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 2.6.4.

**C binding**

`MPI_Comm 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_Comm 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.

`MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)`

`MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)`

```
MPI_Group MPI_Group_f2c(MPI_Fint group)

MPI_Fint MPI_Group_c2f(MPI_Group group)

MPI_Request MPI_Request_f2c(MPI_Fint request)

MPI_Fint MPI_Request_c2f(MPI_Request request)

MPI_File MPI_File_f2c(MPI_Fint file)

MPI_Fint MPI_File_c2f(MPI_File file)

MPI_Win MPI_Win_f2c(MPI_Fint win)

MPI_Fint MPI_Win_c2f(MPI_Win win)

MPI_Op MPI_Op_f2c(MPI_Fint op)

MPI_Fint MPI_Op_c2f(MPI_Op op)

MPI_Info MPI_Info_f2c(MPI_Fint info)

MPI_Fint MPI_Info_c2f(MPI_Info info)

MPI_Errhandler MPI_Errhandler_f2c(MPI_Fint errhandler)

MPI_Fint MPI_Errhandler_c2f(MPI_Errhandler errhandler)

MPI_Message MPI_Message_f2c(MPI_Fint message)

MPI_Fint MPI_Message_c2f(MPI_Message message)
```

**Example 17.11** The example below illustrates how the Fortran MPI function MPI_TYPE_COMMIT can be implemented by wrapping the C MPI function MPI_Type_commit with a C wrapper to do handle conversions. In this example a Fortran-C interface is assumed where a Fortran function is all upper case when referred to from C and arguments are passed by addresses.

```
! FORTRAN PROCEDURE
SUBROUTINE MPI_TYPE_COMMIT(DATATYPE, IERR)
INTEGER :: DATATYPE, IERR
CALL MPI_X_TYPE_COMMIT(DATATYPE, IERR)
RETURN
END

/* C wrapper */

void MPI_X_TYPE_COMMIT(MPI_Fint *f_handle, MPI_Fint *ierr)
{
   MPI_Datatype datatype;

   datatype = MPI_Type_f2c(*f_handle);
   *ierr = (MPI_Fint)MPI_Type_commit(&datatype);
   *f_handle = MPI_Type_c2f(datatype);
   return;
```

```
}
```

The same approach can be used for all other MPI functions. The call to MPI_XXX_f2c (resp. MPI_XXX_c2f) can be omitted when the handle is an OUT (resp. IN) argument, rather than INOUT.

> *Rationale.* The design here provides a convenient solution for the prevalent case, where a C wrapper is used to allow Fortran code to call a C library, or C code to call a Fortran library. The use of C wrappers is much more likely than the use of Fortran wrappers, because it is much more likely that a variable of type INTEGER can be passed to C, than a C handle can be passed to Fortran.
>
> Returning the converted value as a function value rather than through the argument list allows the generation of efficient inlined code when these functions are simple (e.g., the identity). The conversion function in the wrapper does not catch an invalid handle argument. Instead, an invalid handle is passed below to the library function, which, presumably, checks its input arguments. (*End of rationale.*)

### 17.2.5 Status

The following two procedures are provided in C to convert from a Fortran (with the mpi module or mpif.h) status (which is an array of integers) to a C status (which is a structure), and vice versa. The conversion occurs on all the information in status, including that which is hidden. That is, no status information is lost in the conversion.

```
int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)
```

If f_status is a valid Fortran status, but not the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, then MPI_Status_f2c returns in c_status a valid C status with the same content. If f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE, or if f_status is not a valid Fortran status, then the call is erroneous.

The C status has the same source, tag and error code values as the Fortran status, and returns the same answers when queried for count, elements, and cancellation. The conversion function may be called with a Fortran status argument that has an undefined error field, in which case the value of the error field in the C status argument is undefined.

Two global variables of type MPI_Fint*, MPI_F_STATUS_IGNORE and MPI_F_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether f_status is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in the mpi module or mpif.h. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

To do the conversion in the other direction, we have the following:

```
int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)
```

This call converts a C status into a Fortran status, and has a behavior similar to MPI_Status_f2c. That is, the value of c_status must not be either MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE.

> *Advice to users.* There exists no separate conversion function for arrays of statuses, since one can simply loop through the array, converting each status with the routines in Figure 17.1. (*End of advice to users.*)

*Rationale.* The handling of MPI_STATUS_IGNORE is required in order to layer libraries with only a C wrapper: if the Fortran call has passed MPI_STATUS_IGNORE, then the C wrapper must handle this correctly. Note that this constant need not have the same value in Fortran and C. If MPI_Status_f2c were to handle MPI_STATUS_IGNORE, then the type of its result would have to be `MPI_Status**`, which was considered an inferior solution. (*End of rationale.*)

Using the `mpi_f08` Fortran module, a status is declared as `TYPE(MPI_Status)`. The C type `MPI_F08_status` can be used to pass a Fortran `TYPE(MPI_Status)` argument into a C routine. Figure 17.1 illustrates all status conversion routines. Some are only available in C, some in both C and Fortran.

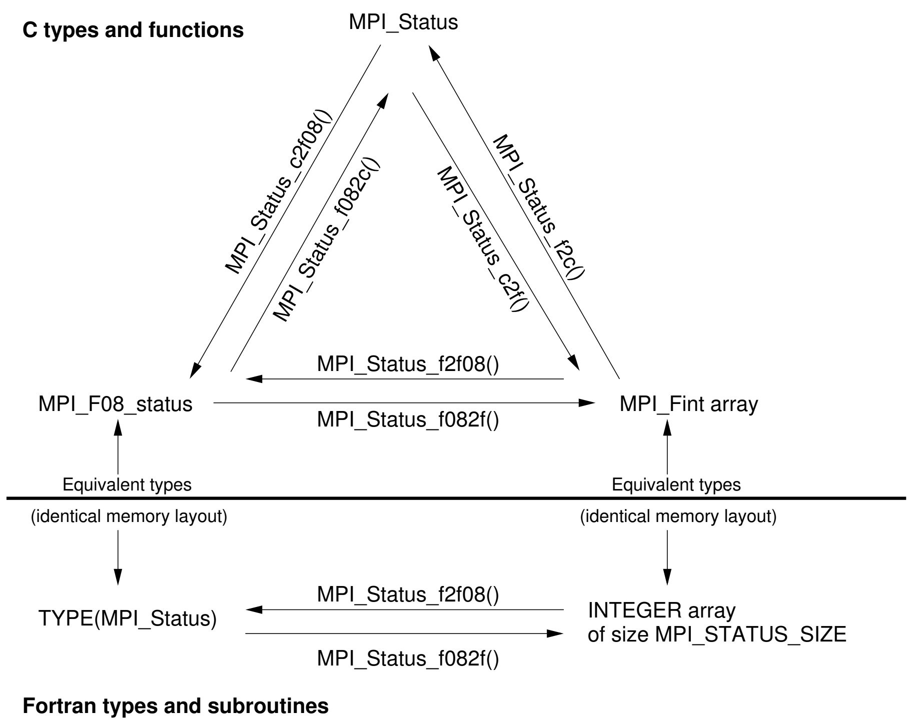


Figure 17.1: Status conversion routines

```
int MPI_Status_f082c(const MPI_F08_status *f08_status,
            MPI_Status *c_status)
```

This C routine converts a Fortran `mpi_f08` `TYPE(MPI_Status)` into a C `MPI_Status`.

```
int MPI_Status_c2f08(const MPI_Status *c_status,
            MPI_F08_status *f08_status)
```

This C routine converts a C `MPI_Status` into a Fortran `mpi_f08` `TYPE(MPI_Status)`. Two global variables of type `MPI_F08_status*`, MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE are declared in mpi.h. They can be used to test, in C, whether `f_status` is the Fortran value of MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE defined in the `mpi_f08` module. These are global variables, not C constant expressions and cannot be used in places where C requires constant expressions. Their value is defined only between the calls to MPI_INIT and MPI_FINALIZE and should not be changed by user code.

Conversion between the two Fortran versions of a status can be done with:

MPI_STATUS_F2F08(f_status, f08_status)

| | | |
|---|---|---|
| IN | f_status | status object declared as array |
| OUT | f08_status | status object declared as named type |

**C binding**

```
int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)
```

**F08 binding**

```
MPI_Status_f2f08(f_status, f08_status, ierror)
    INTEGER, INTENT(IN) :: f_status(MPI_STATUS_SIZE)
    TYPE(MPI_Status), INTENT(OUT) :: f08_status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
    INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
    TYPE(MPI_Status) :: F08_STATUS
```

This routine converts a Fortran `INTEGER, DIMENSION(MPI_STATUS_SIZE)` status array into a Fortran `mpi_f08` `TYPE(MPI_Status)`.

MPI_STATUS_F082F(f08_status, f_status)

| | | |
|---|---|---|
| IN | f08_status | status object declared as named type |
| OUT | f_status | status object declared as array |

**C binding**

```
int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)
```

**F08 binding**

```
MPI_Status_f082f(f08_status, f_status, ierror)
    TYPE(MPI_Status), INTENT(IN) :: f08_status
    INTEGER, INTENT(OUT) :: f_status(MPI_STATUS_SIZE)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

**F binding**

```
MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
    TYPE(MPI_Status) :: F08_STATUS
    INTEGER :: F_STATUS(MPI_STATUS_SIZE), IERROR
```

This routine converts a Fortran `mpi_f08` `TYPE(MPI_Status)` into a Fortran `INTEGER, DIMENSION(MPI_STATUS_SIZE)` status array.

### 17.2.6 MPI Opaque Objects

Unless said otherwise, opaque objects are "the same" in all languages: they carry the same information, and have the same meaning in both languages. The mechanism described in the previous section can be used to pass references to MPI objects from language to language. An object created in one language can be accessed, modified or freed in another language.

We examine below in more detail issues that arise for each type of MPI object.

### Datatypes

Datatypes encode the same information in all languages. E.g., a datatype accessor like MPI_TYPE_GET_EXTENT will return the same information in all languages. If a datatype defined in one language is used for a communication call in another language, then the message sent will be identical to the message that would be sent from the first language: the same communication buffer is accessed, and the same representation conversion is performed, if needed. All predefined datatypes can be used in datatype constructors in any language. If a datatype is committed, it can be used for communication in any language.

The function MPI_GET_ADDRESS returns the same value in all languages. Note that we do not require that the constant MPI_BOTTOM have the same value in all languages (see Section 17.2.9).

**Example 17.12**

```
! FORTRAN CODE
REAL :: R(5)
INTEGER :: TYPE, IERR, AOBLEN(1), AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)

! create an absolute datatype for array R
AOBLEN(1) = 5
CALL MPI_GET_ADDRESS(R, AODISP(1), IERR)
AOTYPE(1) = MPI_REAL
CALL MPI_TYPE_CREATE_STRUCT(1, AOBLEN, AODISP, AOTYPE, TYPE, IERR)
CALL C_ROUTINE(TYPE)

/* C code */

void C_ROUTINE(MPI_Fint *ftype)
{
   int count = 5;
   int lens[2] = {1,1};
   MPI_Aint displs[2];
   MPI_Datatype types[2], newtype;

   /* create an absolute datatype for buffer that consists   */
   /*  of count, followed by R(5)                             */

   MPI_Get_address(&count, &displs[0]);
   displs[1] = 0;
   types[0] = MPI_INT;
   types[1] = MPI_Type_f2c(*ftype);
   MPI_Type_create_struct(2, lens, displs, types, &newtype);
```

```
    MPI_Type_commit(&newtype);

    MPI_Send(MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD);
    /* the message sent contains an int count of 5, followed  */
    /* by the 5 REAL entries of the Fortran array R.          */
}
```

*Advice to implementors.* The following implementation can be used: MPI addresses, as returned by MPI_GET_ADDRESS, will have the same value in all languages. One obvious choice is that MPI addresses be identical to regular addresses. The address is stored in the datatype, when datatypes with absolute addresses are constructed. When a send or receive operation is performed, then addresses stored in a datatype are interpreted as displacements that are all augmented by a base address. This base address is (the address of) buf, or zero, if buf = MPI_BOTTOM. Thus, if MPI_BOTTOM is zero then a send or receive call with buf = MPI_BOTTOM is implemented exactly as a call with a regular buffer argument: in both cases the base address is buf. On the other hand, if MPI_BOTTOM is not zero, then the implementation has to be slightly different. A test is performed to check whether buf = MPI_BOTTOM. If true, then the base address is zero, otherwise it is buf. In particular, if MPI_BOTTOM does not have the same value in Fortran and C, then an additional test for buf = MPI_BOTTOM is needed in at least one of the languages.

It may be desirable to use a value other than zero for MPI_BOTTOM even in C, so as to distinguish it from a NULL pointer. If MPI_BOTTOM = c then one can still avoid the test buf = MPI_BOTTOM, by using the displacement from MPI_BOTTOM, i.e., the regular address - c, as the MPI address returned by MPI_GET_ADDRESS and stored in absolute datatypes. (*End of advice to implementors.*)

#### 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 language support method), and is used to generate the right calling sequence when the callback function is invoked. (*End of advice to implementors.*)

*Advice to users.* If a subroutine written in one language or Fortran support method wants to pass a callback routine including the predefined Fortran functions (e.g., MPI_COMM_NULL_COPY_FN) to another application routine written in another language or Fortran support method, then it must be guaranteed that both routines use the callback interface definition that is defined for the argument when passing the callback to an MPI routine (e.g., MPI_COMM_CREATE_KEYVAL); see also the advice to users on page 302. (*End of advice to users.*)

#### Error Handlers

*Advice to implementors.* Error handlers, have, in C, a variable length argument list. It might be useful to provide to the handler information on the language environment where the error occurred. (*End of advice to implementors.*)

#### Reduce Operations

All predefined named and unnamed datatypes as listed in Section 5.9.2 can be used in the listed predefined operations independent of the programming language from which the MPI routine is called.

*Advice to users.* Reduce operations receive as one of their arguments the datatype of the operands. Thus, one can define "polymorphic" reduce operations that work for C and Fortran datatypes. (*End of advice to users.*)

### 17.2.7 Attributes

Attribute keys can be allocated in one language and freed in another. Similarly, attribute values can be set in one language and accessed in another. To achieve this, attribute keys will be allocated in an integer range that is valid all languages. The same holds true for system-defined attribute values (such as MPI_TAG_UB, MPI_WTIME_IS_GLOBAL, etc.).

Attribute keys declared in one language are associated with copy and delete functions in that language (the functions provided by the MPI_{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" 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 defines attributes arguments to be of type void* in C, and of type INTEGER, in Fortran. On some systems, INTEGERs will have 32 bits, while C pointers will have 64 bits. This is a problem if communicator attributes are used to move information from a Fortran caller to a C callee, or vice-versa.

MPI behaves as if it stores, internally, address sized attributes. If Fortran INTEGERs are smaller, then the (deprecated) Fortran function MPI_ATTR_GET will return the least significant part of the attribute word; the (deprecated) Fortran function MPI_ATTR_PUT will set the least significant part of the attribute word, which will be sign extended to the entire word. (These two functions may be invoked explicitly by user code, or implicitly, by attribute copying callback functions.)

As for addresses, new functions are provided that manipulate Fortran address sized attributes, and have the same functionality as the old functions in C. These functions are described in Section 6.7. Users are encouraged to use these new functions.

MPI supports two types of attributes: address-valued (pointer) attributes, and integer-valued attributes. C attribute functions put and get address-valued attributes. Fortran attribute functions put and get integer-valued attributes. When an integer-valued attribute is accessed from C, then MPI_XXX_get_attr will return the address of (a pointer to) the integer-valued attribute, which is a pointer to MPI_Aint if the attribute was stored with Fortran MPI_XXX_SET_ATTR, and a pointer to int if it was stored with the deprecated Fortran MPI_ATTR_PUT. When an address-valued attribute is accessed from Fortran, then MPI_XXX_GET_ATTR will convert the address into an integer and return the result of this conversion. This conversion is lossless if new style attribute functions are used, and an integer of kind MPI_ADDRESS_KIND is returned. The conversion may cause truncation if deprecated attribute functions are used. In C, the deprecated routines MPI_Attr_put and MPI_Attr_get behave identical to MPI_Comm_set_attr and MPI_Comm_get_attr.

**Example 17.13**

A. Setting an attribute value in C

```
int set_val = 3;
struct foo set_struct;

/* Set a value that is a pointer to an int */

MPI_Comm_set_attr(MPI_COMM_WORLD, keyval1, &set_val);
/* Set a value that is a pointer to a struct */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval2, &set_struct);
/* Set an integer value */
MPI_Comm_set_attr(MPI_COMM_WORLD, keyval3, (void *) 17);
```

B. Reading the attribute value in C

```
int flag, *get_val;
struct foo *get_struct;

/* Upon successful return, get_val == &set_val
   (and therefore *get_val == 3) */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &get_val, &flag);
/* Upon successful return, get_struct == &set_struct */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &get_struct, &flag);
/* Upon successful return, get_val == (void*) 17 */
/*        i.e., (MPI_Aint) get_val == 17 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval3, &get_val, &flag);
```

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

```
LOGICAL FLAG
INTEGER IERR, GET_VAL, GET_STRUCT

! Upon successful return, GET_VAL == &set_val, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct, possibly truncated
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
```

D. Reading the attribute value with Fortran MPI-2 calls

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) GET_VAL, GET_STRUCT

! Upon successful return, GET_VAL == &set_val
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, GET_VAL, FLAG, IERR)
! Upon successful return, GET_STRUCT == &set_struct
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, GET_STRUCT, FLAG, IERR)
! Upon successful return, GET_VAL == 17
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL3, GET_VAL, FLAG, IERR)
```

**Example 17.14** A. Setting an attribute value with the (deprecated) Fortran MPI-1 call

```
INTEGER IERR, VAL
VAL = 7
CALL MPI_ATTR_PUT(MPI_COMM_WORLD, KEYVAL, VAL, IERR)
```

B. Reading the attribute value in C

```
int flag;
int *value;

/* Upon successful return, value points to internal MPI storage and
   *value == (int) 7 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval, &value, &flag);
```

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

```
LOGICAL FLAG
INTEGER IERR, VALUE

! Upon successful return, VALUE == 7
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
```

D. Reading the attribute value with Fortran MPI-2 calls

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE

! Upon successful return, VALUE == 7 (sign extended)
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL, VALUE, FLAG, IERR)
```

**Example 17.15** A. Setting an attribute value via a Fortran MPI-2 call

```
INTEGER IERR
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE1
INTEGER(KIND=MPI_ADDRESS_KIND) VALUE2
VALUE1 = 42
VALUE2 = INT(2, KIND=MPI_ADDRESS_KIND) ** 40

CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, IERR)
CALL MPI_COMM_SET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, IERR)
```

B. Reading the attribute value in C

```
int flag;
MPI_Aint *value1, *value2;

/* Upon successful return, value1 points to internal MPI storage and
   *value1 == 42 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval1, &value1, &flag);
/* Upon successful return, value2 points to internal MPI storage and
   *value2 == 2^40 */
MPI_Comm_get_attr(MPI_COMM_WORLD, keyval2, &value2, &flag);
```

C. Reading the attribute value with (deprecated) Fortran MPI-1 calls

```
LOGICAL FLAG
INTEGER IERR, VALUE1, VALUE2

! Upon successful return, VALUE1 == 42
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
! Upon successful return, VALUE2 == 2^40, or 0 if truncation
! needed (i.e., the least significant part of the attribute word)
CALL MPI_ATTR_GET(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

D. Reading the attribute value with Fortran MPI-2 calls

```
LOGICAL FLAG
INTEGER IERR
INTEGER (KIND=MPI_ADDRESS_KIND) VALUE1, VALUE2

! Upon successful return, VALUE1 == 42
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL1, VALUE1, FLAG, IERR)
! Upon successful return, VALUE2 == 2^40
CALL MPI_COMM_GET_ATTR(MPI_COMM_WORLD, KEYVAL2, VALUE2, FLAG, IERR)
```

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 call to the deprecated Fortran routine MPI_ATTR_PUT, i.e., in Fortran, MPI_COMM_GET_ATTR(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, ierror) will return in val the base address of the window, converted to an integer. In C, MPI_Win_get_attr(win, MPI_WIN_BASE, &p, &flag) will return in p a pointer to the window base, cast to (void *).

*Rationale.* The design is consistent with the behavior specified for predefined attributes, and ensures that no information is lost when attributes are passed from language to language. Because the language interoperability for predefined attributes was defined based on MPI_ATTR_PUT, this definition is kept for compatibility reasons although the routine itself is now deprecated. (*End of rationale.*)

*Advice to implementors.* Implementations should tag attributes either as (1) address attributes, (2) as INTEGER(KIND=MPI_ADDRESS_KIND) attributes or (3) as INTEGER attributes, according to whether they were set in (1) C (with MPI_Attr_put or MPI_XXX_set_attr), (2) in Fortran with MPI_XXX_SET_ATTR or (3) with the deprecated Fortran routine MPI_ATTR_PUT. Thus, the right choice can be made when the attribute is retrieved. (*End of advice to implementors.*)

### 17.2.8 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.

### 17.2.9 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 17.2.4. Constants that specify maximum lengths of strings (see Section A.1.1 for a listing) have a value one less in Fortran than C since in C the length includes the null terminating character. Thus, these constants represent the amount of space which must be allocated to hold the largest possible such string, rather than the maximum number of printable characters the string could contain.

*Advice to users.* This definition means that it is safe in C to allocate a buffer to receive a string using a declaration like

```
char name [MPI_MAX_OBJECT_NAME];
```

(*End of advice to users.*)

Also constant "addresses," i.e., special values for reference arguments that are not handles, such as MPI_BOTTOM or MPI_STATUS_IGNORE may have different values in different languages.

*Rationale.* The current MPI standard specifies that MPI_BOTTOM can be used in initialization expressions in C, but not in Fortran. Since Fortran does not normally support call by value, then MPI_BOTTOM in Fortran must be the name of a predefined static variable, e.g., a variable in an MPI declared COMMON block. On the other hand, in C, it is natural to take MPI_BOTTOM = 0 (Caveat: Defining MPI_BOTTOM = 0 implies that NULL pointer cannot be distinguished from MPI_BOTTOM; it may be that MPI_BOTTOM = 1 is better. See the advice to implementors in the *Datatypes* subsection in Section 17.2.6) Requiring that the Fortran and C values be the same will complicate the initialization process. (*End of rationale.*)

### 17.2.10 Interlanguage Communication

The type matching rules for communication in MPI are not changed: the datatype specification for each item sent should match, in type signature, the datatype specification used to receive this item (unless one of the types is MPI_PACKED). Also, the type of a message item should match the type declaration for the corresponding communication buffer location, unless the type is MPI_BYTE or MPI_PACKED. Interlanguage communication is allowed if it complies with these rules.

**Example 17.16** In the example below, a Fortran array is sent from Fortran and received in C.

```
! FORTRAN CODE
SUBROUTINE MYEXAMPLE()
USE mpi_f08
REAL :: R(5)
INTEGER :: IERR, MYRANK, AOBLEN(1)
TYPE(MPI_Datatype) :: TYPE, AOTYPE(1)
INTEGER (KIND=MPI_ADDRESS_KIND) :: AODISP(1)

! create an absolute datatype for array R
AOBLEN(1) = 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%MPI_VAL)
END IF
END SUBROUTINE
```

```
/* C code */

void C_ROUTINE(MPI_Fint *fhandle)
{
   MPI_Datatype type;
   MPI_Status status;

   type = MPI_Type_f2c(*fhandle);

   MPI_Recv(MPI_BOTTOM, 1, type, 0, 0, MPI_COMM_WORLD, &status);
}
```

MPI implementors may weaken these type matching rules, and allow messages to be sent with Fortran types and received with C types, and vice versa, when those types match. I.e., 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 and Fortran. First we present the constants, type definitions, info values and keys. Then we present the routine prototypes separately for each binding. Listings are alphabetical within chapter.

## A.1 Defined Values and Handles

### A.1.1 Defined Constants

The C and Fortran names are listed below. Constants with the type `const int` may also be implemented as literal integer constants substituted by the preprocessor.

**Error classes**

| C type: `const int` (or unnamed `enum`)<br>Fortran type: `INTEGER` |
|---|
| MPI_SUCCESS |
| MPI_ERR_BUFFER |
| MPI_ERR_COUNT |
| MPI_ERR_TYPE |
| MPI_ERR_TAG |
| MPI_ERR_COMM |
| MPI_ERR_RANK |
| MPI_ERR_REQUEST |
| MPI_ERR_ROOT |
| MPI_ERR_GROUP |
| MPI_ERR_OP |
| MPI_ERR_TOPOLOGY |
| MPI_ERR_DIMS |
| MPI_ERR_ARG |
| MPI_ERR_UNKNOWN |
| MPI_ERR_TRUNCATE |
| MPI_ERR_OTHER |
| MPI_ERR_INTERN |
| MPI_ERR_PENDING |

**(Continued on next page)**

| Error classes (continued) |
| --- |
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_ERR_IN_STATUS |
| MPI_ERR_ACCESS |
| MPI_ERR_AMODE |
| MPI_ERR_ASSERT |
| MPI_ERR_BAD_FILE |
| MPI_ERR_BASE |
| MPI_ERR_CONVERSION |
| MPI_ERR_DISP |
| MPI_ERR_DUP_DATAREP |
| MPI_ERR_FILE_EXISTS |
| MPI_ERR_FILE_IN_USE |
| MPI_ERR_FILE |
| MPI_ERR_INFO_KEY |
| MPI_ERR_INFO_NOKEY |
| MPI_ERR_INFO_VALUE |
| MPI_ERR_INFO |
| MPI_ERR_IO |
| MPI_ERR_KEYVAL |
| MPI_ERR_LOCKTYPE |
| MPI_ERR_NAME |
| MPI_ERR_NO_MEM |
| MPI_ERR_NOT_SAME |
| MPI_ERR_NO_SPACE |
| MPI_ERR_NO_SUCH_FILE |
| MPI_ERR_PORT |
| MPI_ERR_QUOTA |
| MPI_ERR_READ_ONLY |
| MPI_ERR_RMA_ATTACH |
| MPI_ERR_RMA_CONFLICT |
| MPI_ERR_RMA_RANGE |
| MPI_ERR_RMA_SHARED |
| MPI_ERR_RMA_SYNC |
| MPI_ERR_RMA_FLAVOR |
| MPI_ERR_SERVICE |
| MPI_ERR_SIZE |
| MPI_ERR_SPAWN |
| MPI_ERR_UNSUPPORTED_DATAREP |
| MPI_ERR_UNSUPPORTED_OPERATION |
| MPI_ERR_WIN |

**(Continued on next page)**

| Error classes (continued) |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_T_ERR_CANNOT_INIT |
| MPI_T_ERR_NOT_INITIALIZED |
| MPI_T_ERR_MEMORY |
| MPI_T_ERR_INVALID |
| MPI_T_ERR_INVALID_INDEX |
| MPI_T_ERR_INVALID_ITEM |
| MPI_T_ERR_INVALID_SESSION |
| MPI_T_ERR_INVALID_HANDLE |
| MPI_T_ERR_INVALID_NAME |
| MPI_T_ERR_OUT_OF_HANDLES |
| MPI_T_ERR_OUT_OF_SESSIONS |
| MPI_T_ERR_CVAR_SET_NOT_NOW |
| MPI_T_ERR_CVAR_SET_NEVER |
| MPI_T_ERR_PVAR_NO_WRITE |
| MPI_T_ERR_PVAR_NO_STARTSTOP |
| MPI_T_ERR_PVAR_NO_ATOMIC |
| MPI_ERR_LASTCODE |

| Buffer Address Constants |
|---|
| C type: `void * const` |
| Fortran type: (predefined memory location)[1] |
| MPI_BOTTOM |
| MPI_IN_PLACE |

[1] Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.

| Assorted Constants |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_PROC_NULL |
| MPI_ANY_SOURCE |
| MPI_ANY_TAG |
| MPI_UNDEFINED |
| MPI_BSEND_OVERHEAD |
| MPI_KEYVAL_INVALID |
| MPI_LOCK_EXCLUSIVE |
| MPI_LOCK_SHARED |
| MPI_ROOT |

| No Process Message Handle |
|---|
| C type: `MPI_Message` |
| Fortran type: `INTEGER` or `TYPE(MPI_Message)` |
| MPI_MESSAGE_NO_PROC |

| Fortran Support Method Specific Constants |
|---|
| Fortran type: LOGICAL |
| MPI_SUBARRAYS_SUPPORTED (Fortran only) |
| MPI_ASYNC_PROTECTS_NONBLOCKING (Fortran only) |

| Status size and reserved index values (Fortran only) |
|---|
| Fortran type: INTEGER |
| MPI_STATUS_SIZE |
| MPI_SOURCE |
| MPI_TAG |
| MPI_ERROR |

| Variable Address Size (Fortran only) |
|---|
| Fortran type: INTEGER |
| MPI_ADDRESS_KIND |
| MPI_COUNT_KIND |
| MPI_INTEGER_KIND |
| MPI_OFFSET_KIND |

| Error-handling specifiers |
|---|
| C type: MPI_Errhandler<br>Fortran type: INTEGER or TYPE(MPI_Errhandler) |
| MPI_ERRORS_ARE_FATAL |
| MPI_ERRORS_RETURN |

| Maximum Sizes for Strings |
|---|
| C type: const int (or unnamed enum)<br>Fortran type: INTEGER |
| MPI_MAX_DATAREP_STRING |
| MPI_MAX_ERROR_STRING |
| MPI_MAX_INFO_KEY |
| MPI_MAX_INFO_VAL |
| MPI_MAX_LIBRARY_VERSION_STRING |
| MPI_MAX_OBJECT_NAME |
| MPI_MAX_PORT_NAME |
| MPI_MAX_PROCESSOR_NAME |

| **Named Predefined Datatypes** | **C types** |
|---|---|
| C type: `MPI_Datatype`<br>Fortran type: `INTEGER`<br>or `TYPE(MPI_Datatype)` | |
| MPI_CHAR | `char`<br>(treated as printable character) |
| MPI_SHORT | `signed short int` |
| MPI_INT | `signed int` |
| MPI_LONG | `signed long` |
| MPI_LONG_LONG_INT | `signed long long` |
| MPI_LONG_LONG (as a synonym) | `signed long long` |
| MPI_SIGNED_CHAR | `signed char`<br>(treated as integral value) |
| MPI_UNSIGNED_CHAR | `unsigned char`<br>(treated as integral value) |
| MPI_UNSIGNED_SHORT | `unsigned short` |
| MPI_UNSIGNED | `unsigned int` |
| MPI_UNSIGNED_LONG | `unsigned long` |
| MPI_UNSIGNED_LONG_LONG | `unsigned long long` |
| MPI_FLOAT | `float` |
| MPI_DOUBLE | `double` |
| MPI_LONG_DOUBLE | `long double` |
| MPI_WCHAR | `wchar_t`<br>(defined in `<stddef.h>`)<br>(treated as printable character) |
| MPI_C_BOOL | `_Bool` |
| MPI_INT8_T | `int8_t` |
| MPI_INT16_T | `int16_t` |
| MPI_INT32_T | `int32_t` |
| MPI_INT64_T | `int64_t` |
| MPI_UINT8_T | `uint8_t` |
| MPI_UINT16_T | `uint16_t` |
| MPI_UINT32_T | `uint32_t` |
| MPI_UINT64_T | `uint64_t` |
| MPI_AINT | `MPI_Aint` |
| MPI_COUNT | `MPI_Count` |
| MPI_OFFSET | `MPI_Offset` |
| MPI_C_COMPLEX | `float _Complex` |
| MPI_C_FLOAT_COMPLEX | `float _Complex` |
| MPI_C_DOUBLE_COMPLEX | `double _Complex` |
| MPI_C_LONG_DOUBLE_COMPLEX | `long double _Complex` |
| MPI_BYTE | (any C type) |
| MPI_PACKED | (any C type) |

| **Named Predefined Datatypes** | **Fortran types** |
|---|---|
| C type: MPI_Datatype<br>Fortran type: INTEGER<br>or TYPE(MPI_Datatype) | |
| MPI_INTEGER | INTEGER |
| MPI_REAL | REAL |
| MPI_DOUBLE_PRECISION | DOUBLE PRECISION |
| MPI_COMPLEX | COMPLEX |
| MPI_LOGICAL | LOGICAL |
| MPI_CHARACTER | CHARACTER(1) |
| MPI_AINT | INTEGER (KIND=MPI_ADDRESS_KIND) |
| MPI_COUNT | INTEGER (KIND=MPI_COUNT_KIND) |
| MPI_OFFSET | INTEGER (KIND=MPI_OFFSET_KIND) |
| MPI_BYTE | (any Fortran type) |
| MPI_PACKED | (any Fortran type) |

| **Named Predefined Datatypes**[1] | **C++ types** |
|---|---|
| C type: MPI_Datatype<br>Fortran type: INTEGER<br>or TYPE(MPI_Datatype) | |
| MPI_CXX_BOOL | bool |
| MPI_CXX_FLOAT_COMPLEX | std::complex<float> |
| MPI_CXX_DOUBLE_COMPLEX | std::complex<double> |
| MPI_CXX_LONG_DOUBLE_COMPLEX | std::complex<long double> |

[1] If an accompanying C++ compiler is missing, then the MPI datatypes in this table are not defined.

| **Optional datatypes (Fortran)** | **Fortran types** |
|---|---|
| C type: MPI_Datatype<br>Fortran type: INTEGER<br>or TYPE(MPI_Datatype) | |
| MPI_DOUBLE_COMPLEX | DOUBLE COMPLEX |
| MPI_INTEGER1 | INTEGER*1 |
| MPI_INTEGER2 | INTEGER*2 |
| MPI_INTEGER4 | INTEGER*4 |
| MPI_INTEGER8 | INTEGER*8 |
| MPI_INTEGER16 | INTEGER*16 |
| MPI_REAL2 | REAL*2 |
| MPI_REAL4 | REAL*4 |
| MPI_REAL8 | REAL*8 |
| MPI_REAL16 | REAL*16 |
| MPI_COMPLEX4 | COMPLEX*4 |
| MPI_COMPLEX8 | COMPLEX*8 |
| MPI_COMPLEX16 | COMPLEX*16 |
| MPI_COMPLEX32 | COMPLEX*32 |

| **Datatypes for reduction functions (C)** |
|---|
| C type: `MPI_Datatype` |
| Fortran type: `INTEGER` or `TYPE(MPI_Datatype)` |
| MPI_FLOAT_INT |
| MPI_DOUBLE_INT |
| MPI_LONG_INT |
| MPI_2INT |
| MPI_SHORT_INT |
| MPI_LONG_DOUBLE_INT |

| **Datatypes for reduction functions (Fortran)** |
|---|
| C type: `MPI_Datatype` |
| Fortran type: `INTEGER` or `TYPE(MPI_Datatype)` |
| MPI_2REAL |
| MPI_2DOUBLE_PRECISION |
| MPI_2INTEGER |

| **Reserved communicators** |
|---|
| C type: `MPI_Comm` |
| Fortran type: `INTEGER` or `TYPE(MPI_Comm)` |
| MPI_COMM_WORLD |
| MPI_COMM_SELF |

| **Communicator split type constants** |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_COMM_TYPE_SHARED |

| **Results of communicator and group comparisons** |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_IDENT |
| MPI_CONGRUENT |
| MPI_SIMILAR |
| MPI_UNEQUAL |

| **Environmental inquiry info key** |
|---|
| C type: `MPI_Info` |
| Fortran type: `INTEGER` or `TYPE(MPI_Info)` |
| MPI_INFO_ENV |

| **Environmental inquiry keys** |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_TAG_UB |
| MPI_IO |
| MPI_HOST |
| MPI_WTIME_IS_GLOBAL |

**Collective Operations**

| C type: MPI_Op<br>Fortran type: INTEGER or TYPE(MPI_Op) |
|---|
| MPI_MAX |
| MPI_MIN |
| MPI_SUM |
| MPI_PROD |
| MPI_MAXLOC |
| MPI_MINLOC |
| MPI_BAND |
| MPI_BOR |
| MPI_BXOR |
| MPI_LAND |
| MPI_LOR |
| MPI_LXOR |
| MPI_REPLACE |
| MPI_NO_OP |

**Null Handles**

| C/Fortran name<br>C type / Fortran type |
|---|
| MPI_GROUP_NULL<br>MPI_Group / INTEGER or TYPE(MPI_Group) |
| MPI_COMM_NULL<br>MPI_Comm / INTEGER or TYPE(MPI_Comm) |
| MPI_DATATYPE_NULL<br>MPI_Datatype / INTEGER or TYPE(MPI_Datatype) |
| MPI_REQUEST_NULL<br>MPI_Request / INTEGER or TYPE(MPI_Request) |
| MPI_OP_NULL<br>MPI_Op / INTEGER or TYPE(MPI_Op) |
| MPI_ERRHANDLER_NULL<br>MPI_Errhandler / INTEGER or TYPE(MPI_Errhandler) |
| MPI_FILE_NULL<br>MPI_File / INTEGER or TYPE(MPI_File) |
| MPI_INFO_NULL<br>MPI_Info / INTEGER or TYPE(MPI_Info) |
| MPI_WIN_NULL<br>MPI_Win / INTEGER or TYPE(MPI_Win) |
| MPI_MESSAGE_NULL<br>MPI_Message / INTEGER or TYPE(MPI_Message) |

**Empty group**

| C type: MPI_Group<br>Fortran type: INTEGER or TYPE(MPI_Group) |
|---|
| MPI_GROUP_EMPTY |

**Topologies**

| C type: `const int` (or unnamed `enum`)<br>Fortran type: `INTEGER` |
|---|
| MPI_GRAPH |
| MPI_CART |
| MPI_DIST_GRAPH |

**Predefined functions**

| C/Fortran name<br>C type<br>/ Fortran type with `mpi` module / Fortran type with `mpi_f08` module |
|---|
| MPI_COMM_NULL_COPY_FN<br>`MPI_Comm_copy_attr_function`<br>/ `COMM_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Comm_copy_attr_function)` [1]) |
| MPI_COMM_DUP_FN<br>`MPI_Comm_copy_attr_function`<br>/ `COMM_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Comm_copy_attr_function)` [1]) |
| MPI_COMM_NULL_DELETE_FN<br>`MPI_Comm_delete_attr_function`<br>/ `COMM_DELETE_ATTR_FUNCTION` / `PROCEDURE(MPI_Comm_delete_attr_function)` [1]) |
| MPI_WIN_NULL_COPY_FN<br>`MPI_Win_copy_attr_function`<br>/ `WIN_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Win_copy_attr_function)` [1]) |
| MPI_WIN_DUP_FN<br>`MPI_Win_copy_attr_function`<br>/ `WIN_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Win_copy_attr_function)` [1]) |
| MPI_WIN_NULL_DELETE_FN<br>`MPI_Win_delete_attr_function`<br>/ `WIN_DELETE_ATTR_FUNCTION` / `PROCEDURE(MPI_Win_delete_attr_function)` [1]) |
| MPI_TYPE_NULL_COPY_FN<br>`MPI_Type_copy_attr_function`<br>/ `TYPE_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Type_copy_attr_function)` [1]) |
| MPI_TYPE_DUP_FN<br>`MPI_Type_copy_attr_function`<br>/ `TYPE_COPY_ATTR_FUNCTION` / `PROCEDURE(MPI_Type_copy_attr_function)` [1]) |
| MPI_TYPE_NULL_DELETE_FN<br>`MPI_Type_delete_attr_function`<br>/ `TYPE_DELETE_ATTR_FUNCTION` / `PROCEDURE(MPI_Type_delete_attr_function)` [1]) |
| MPI_CONVERSION_FN_NULL<br>`MPI_Datarep_conversion_function`<br>/ `DATAREP_CONVERSION_FUNCTION` / `PROCEDURE(MPI_Datarep_conversion_function)` [1]) |
| [1] See the advice to implementors (on page 301) and advice to users (on page 302) on the predefined Fortran functions MPI_COMM_NULL_COPY_FN, ... in Section 6.7.2. |

**Deprecated predefined functions**

| C/Fortran name<br>C type / Fortran type with mpi module |
|---|
| MPI_NULL_COPY_FN<br>MPI_Copy_function / COPY_FUNCTION |
| MPI_DUP_FN<br>MPI_Copy_function / COPY_FUNCTION |
| MPI_NULL_DELETE_FN<br>MPI_Delete_function / DELETE_FUNCTION |

**Predefined Attribute Keys**

| C type: const int (or unnamed enum)<br>Fortran type: INTEGER |
|---|
| MPI_APPNUM |
| MPI_LASTUSEDCODE |
| MPI_UNIVERSE_SIZE |
| MPI_WIN_BASE |
| MPI_WIN_DISP_UNIT |
| MPI_WIN_SIZE |
| MPI_WIN_CREATE_FLAVOR |
| MPI_WIN_MODEL |

**MPI Window Create Flavors**

| C type: const int (or unnamed enum)<br>Fortran type: INTEGER |
|---|
| MPI_WIN_FLAVOR_CREATE |
| MPI_WIN_FLAVOR_ALLOCATE |
| MPI_WIN_FLAVOR_DYNAMIC |
| MPI_WIN_FLAVOR_SHARED |

**MPI Window Models**

| C type: const int (or unnamed enum)<br>Fortran type: INTEGER |
|---|
| MPI_WIN_SEPARATE |
| MPI_WIN_UNIFIED |

| Mode Constants |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_MODE_APPEND |
| MPI_MODE_CREATE |
| MPI_MODE_DELETE_ON_CLOSE |
| MPI_MODE_EXCL |
| MPI_MODE_NOCHECK |
| MPI_MODE_NOPRECEDE |
| MPI_MODE_NOPUT |
| MPI_MODE_NOSTORE |
| MPI_MODE_NOSUCCEED |
| MPI_MODE_RDONLY |
| MPI_MODE_RDWR |
| MPI_MODE_SEQUENTIAL |
| MPI_MODE_UNIQUE_OPEN |
| MPI_MODE_WRONLY |

| Datatype Decoding Constants |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_COMBINER_CONTIGUOUS |
| MPI_COMBINER_DARRAY |
| MPI_COMBINER_DUP |
| MPI_COMBINER_F90_COMPLEX |
| MPI_COMBINER_F90_INTEGER |
| MPI_COMBINER_F90_REAL |
| MPI_COMBINER_HINDEXED |
| MPI_COMBINER_HVECTOR |
| MPI_COMBINER_INDEXED_BLOCK |
| MPI_COMBINER_HINDEXED_BLOCK |
| MPI_COMBINER_INDEXED |
| MPI_COMBINER_NAMED |
| MPI_COMBINER_RESIZED |
| MPI_COMBINER_STRUCT |
| MPI_COMBINER_SUBARRAY |
| MPI_COMBINER_VECTOR |

| Threads Constants |
|---|
| C type: `const int` (or unnamed `enum`) |
| Fortran type: `INTEGER` |
| MPI_THREAD_FUNNELED |
| MPI_THREAD_MULTIPLE |
| MPI_THREAD_SERIALIZED |
| MPI_THREAD_SINGLE |

| File Operation Constants, Part 1 |
|---|
| C type: const MPI_Offset (or unnamed enum)<br>Fortran type: INTEGER (KIND=MPI_OFFSET_KIND) |
| MPI_DISPLACEMENT_CURRENT |

| File Operation Constants, Part 2 |
|---|
| C type: const int (or unnamed enum)<br>Fortran type: INTEGER |
| MPI_DISTRIBUTE_BLOCK |
| MPI_DISTRIBUTE_CYCLIC |
| MPI_DISTRIBUTE_DFLT_DARG |
| MPI_DISTRIBUTE_NONE |
| MPI_ORDER_C |
| MPI_ORDER_FORTRAN |
| MPI_SEEK_CUR |
| MPI_SEEK_END |
| MPI_SEEK_SET |

| F90 Datatype Matching Constants |
|---|
| C type: const int (or unnamed enum)<br>Fortran type: INTEGER |
| MPI_TYPECLASS_COMPLEX |
| MPI_TYPECLASS_INTEGER |
| MPI_TYPECLASS_REAL |

| Constants Specifying Empty or Ignored Input |
|---|
| C/Fortran name<br>C type / Fortran type[1] |
| MPI_ARGVS_NULL<br>char*** / 2-dim. array of CHARACTER*(*) |
| MPI_ARGV_NULL<br>char** / array of CHARACTER*(*) |
| MPI_ERRCODES_IGNORE<br>int* / INTEGER array |
| MPI_STATUSES_IGNORE<br>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE,*)<br>or TYPE(MPI_Status), DIMENSION(*) |
| MPI_STATUS_IGNORE<br>MPI_Status* / INTEGER, DIMENSION(MPI_STATUS_SIZE)<br>or TYPE(MPI_Status) |
| MPI_UNWEIGHTED<br>int* / INTEGER array |
| MPI_WEIGHTS_EMPTY<br>int* / INTEGER array |

[1] Note that in Fortran these constants are not usable for initialization expressions or assignment. See Section 2.5.4.

**C Constants Specifying Ignored Input (no Fortran)**

| C type: MPI_Fint* | equivalent to Fortran |
|---|---|
| MPI_F_STATUSES_IGNORE | MPI_STATUSES_IGNORE in mpi / mpif.h |
| MPI_F_STATUS_IGNORE | MPI_STATUS_IGNORE in mpi / mpif.h |
| C type: MPI_F08_status* | equivalent to Fortran |
| MPI_F08_STATUSES_IGNORE | MPI_STATUSES_IGNORE in mpi_f08 |
| MPI_F08_STATUS_IGNORE | MPI_STATUS_IGNORE in mpi_f08 |

**C preprocessor Constants and Fortran Parameters**

| C type: C-preprocessor macro that expands to an int value<br>Fortran type: INTEGER |
|---|
| MPI_SUBVERSION |
| MPI_VERSION |

**Null handles used in the MPI tool information interface**

| |
|---|
| MPI_T_ENUM_NULL |
| MPI_T_enum |
| MPI_T_CVAR_HANDLE_NULL |
| MPI_T_cvar_handle |
| MPI_T_PVAR_HANDLE_NULL |
| MPI_T_pvar_handle |
| MPI_T_PVAR_SESSION_NULL |
| MPI_T_pvar_session |

**Verbosity Levels in the MPI tool information interface**

| C type: const int (or unnamed enum) |
|---|
| MPI_T_VERBOSITY_USER_BASIC |
| MPI_T_VERBOSITY_USER_DETAIL |
| MPI_T_VERBOSITY_USER_ALL |
| MPI_T_VERBOSITY_TUNER_BASIC |
| MPI_T_VERBOSITY_TUNER_DETAIL |
| MPI_T_VERBOSITY_TUNER_ALL |
| MPI_T_VERBOSITY_MPIDEV_BASIC |
| MPI_T_VERBOSITY_MPIDEV_DETAIL |
| MPI_T_VERBOSITY_MPIDEV_ALL |

| Constants to identify associations of variables in the MPI tool information interface |
|---|
| C type: `const int` (or unnamed `enum`) |
| MPI_T_BIND_NO_OBJECT |
| MPI_T_BIND_MPI_COMM |
| MPI_T_BIND_MPI_DATATYPE |
| MPI_T_BIND_MPI_ERRHANDLER |
| MPI_T_BIND_MPI_FILE |
| MPI_T_BIND_MPI_GROUP |
| MPI_T_BIND_MPI_OP |
| MPI_T_BIND_MPI_REQUEST |
| MPI_T_BIND_MPI_WIN |
| MPI_T_BIND_MPI_MESSAGE |
| MPI_T_BIND_MPI_INFO |

| Constants describing the scope of a control variable in the MPI tool information interface |
|---|
| C type: `const int` (or unnamed `enum`) |
| MPI_T_SCOPE_CONSTANT |
| MPI_T_SCOPE_READONLY |
| MPI_T_SCOPE_LOCAL |
| MPI_T_SCOPE_GROUP |
| MPI_T_SCOPE_GROUP_EQ |
| MPI_T_SCOPE_ALL |
| MPI_T_SCOPE_ALL_EQ |

| Additional constants used by the MPI tool information interface |
|---|
| C type: `MPI_T_pvar_handle` |
| MPI_T_PVAR_ALL_HANDLES |

| Performance variables classes used by the MPI tool information interface |
|---|
| C type: `const int` (or unnamed `enum`) |
| MPI_T_PVAR_CLASS_STATE |
| MPI_T_PVAR_CLASS_LEVEL |
| MPI_T_PVAR_CLASS_SIZE |
| MPI_T_PVAR_CLASS_PERCENTAGE |
| MPI_T_PVAR_CLASS_HIGHWATERMARK |
| MPI_T_PVAR_CLASS_LOWWATERMARK |
| MPI_T_PVAR_CLASS_COUNTER |
| MPI_T_PVAR_CLASS_AGGREGATE |
| MPI_T_PVAR_CLASS_TIMER |
| MPI_T_PVAR_CLASS_GENERIC |

### A.1.2 Types

The following are defined C type definitions, included in the file `mpi.h`.

```
/* C opaque types */
MPI_Aint
MPI_Count
MPI_Fint
MPI_Offset
MPI_Status
MPI_F08_status

/* C handles to assorted structures */
MPI_Comm
MPI_Datatype
MPI_Errhandler
MPI_File
MPI_Group
MPI_Info
MPI_Message
MPI_Op
MPI_Request
MPI_Win

/* Types for the MPI_T interface */
MPI_T_enum
MPI_T_cvar_handle
MPI_T_pvar_handle
MPI_T_pvar_session
```

The following are defined Fortran type definitions, included in the mpi_f08 and mpi modules.

```
!  Fortran opaque types in the mpi_f08 and mpi modules
TYPE(MPI_Status)

!  Fortran handles in the mpi_f08 and mpi modules
TYPE(MPI_Comm)
TYPE(MPI_Datatype)
TYPE(MPI_Errhandler)
TYPE(MPI_File)
TYPE(MPI_Group)
TYPE(MPI_Info)
TYPE(MPI_Message)
TYPE(MPI_Op)
TYPE(MPI_Request)
TYPE(MPI_Win)
```

### A.1.3 Prototype Definitions

#### C Bindings

The following are defined C typedefs for user-defined functions, also included in the file mpi.h.

```
/* prototypes for user-defined functions */
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
              MPI_Datatype *datatype);

typedef int MPI_Comm_copy_attr_function(MPI_Comm oldcomm,
              int comm_keyval, void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
typedef int MPI_Comm_delete_attr_function(MPI_Comm comm,
              int comm_keyval, void *attribute_val, void *extra_state);

typedef int MPI_Win_copy_attr_function(MPI_Win oldwin, int win_keyval,
              void *extra_state, void *attribute_val_in,
              void *attribute_val_out, int *flag);
typedef int MPI_Win_delete_attr_function(MPI_Win win, int win_keyval,
              void *attribute_val, void *extra_state);

typedef int MPI_Type_copy_attr_function(MPI_Datatype oldtype,
              int type_keyval, void *extra_state,
              void *attribute_val_in, void *attribute_val_out, int *flag);
typedef int MPI_Type_delete_attr_function(MPI_Datatype datatype,
              int type_keyval, void *attribute_val, void *extra_state);

typedef void MPI_Comm_errhandler_function(MPI_Comm *, int *, ...);
typedef void MPI_Win_errhandler_function(MPI_Win *, int *, ...);
typedef void MPI_File_errhandler_function(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);
```

#### Fortran 2008 Bindings with the mpi_f08 Module

The callback prototypes when using the Fortran mpi_f08 module are shown below:

The user-function argument to MPI_Op_create should be declared according to:

```
ABSTRACT INTERFACE
```

```
  SUBROUTINE MPI_User_function(invec, inoutvec, len, datatype)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE :: invec, inoutvec
      INTEGER :: len
      TYPE(MPI_Datatype) :: datatype
```

The copy and delete function arguments to MPI_Comm_create_keyval should be declared according to:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_copy_attr_function(oldcomm, comm_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Comm) :: oldcomm
      INTEGER :: comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL :: flag

ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_delete_attr_function(comm, comm_keyval,
  attribute_val, extra_state, ierror)
      TYPE(MPI_Comm) :: comm
      INTEGER :: comm_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

The copy and delete function arguments to MPI_Win_create_keyval should be declared according to:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_copy_attr_function(oldwin, win_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Win) :: oldwin
      INTEGER :: win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
      attribute_val_out
      LOGICAL :: flag

ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_delete_attr_function(win, win_keyval, attribute_val,
  extra_state, ierror)
      TYPE(MPI_Win) :: win
      INTEGER :: win_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

The copy and delete function arguments to MPI_Type_create_keyval should be declared according to:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_copy_attr_function(oldtype, type_keyval, extra_state,
  attribute_val_in, attribute_val_out, flag, ierror)
      TYPE(MPI_Datatype) :: oldtype
      INTEGER :: type_keyval, ierror
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in,
```

```
        attribute_val_out
        LOGICAL :: flag

ABSTRACT INTERFACE
  SUBROUTINE MPI_Type_delete_attr_function(datatype, type_keyval,
  attribute_val, extra_state, ierror)
        TYPE(MPI_Datatype) :: datatype
        INTEGER :: type_keyval, ierror
        INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
```

The handler-function argument to MPI_Comm_create_errhandler should be declared like this:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Comm_errhandler_function(comm, error_code)
        TYPE(MPI_Comm) :: comm
        INTEGER :: error_code
```

The handler-function argument to MPI_Win_create_errhandler should be declared like this:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Win_errhandler_function(win, error_code)
        TYPE(MPI_Win) :: win
        INTEGER :: error_code
```

The handler-function argument to MPI_File_create_errhandler should be declared like this:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_File_errhandler_function(file, error_code)
        TYPE(MPI_File) :: file
        INTEGER :: error_code
```

The query, free, and cancel function arguments to MPI_Grequest_start should be declared according to:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_query_function(extra_state, status, ierror)
        TYPE(MPI_Status) :: status
        INTEGER :: ierror
        INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state

ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_free_function(extra_state, ierror)
        INTEGER :: ierror
        INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state

ABSTRACT INTERFACE
  SUBROUTINE MPI_Grequest_cancel_function(extra_state, complete, ierror)
        INTEGER :: ierror
        INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
        LOGICAL :: complete
```

The extent and conversion function arguments to MPI_Register_datarep should be declared according to:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_extent_function(datatype, extent, extra_state,
  ierror)
      TYPE(MPI_Datatype) :: datatype
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extent, extra_state
      INTEGER :: ierror

ABSTRACT INTERFACE
  SUBROUTINE MPI_Datarep_conversion_function(userbuf, datatype, count,
  filebuf, position, extra_state, ierror)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
      TYPE(C_PTR), VALUE :: userbuf, filebuf
      TYPE(MPI_Datatype) :: datatype
      INTEGER :: count, ierror
      INTEGER(KIND=MPI_OFFSET_KIND) :: position
      INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state
```

#### Fortran Bindings with mpif.h or the mpi Module

With the Fortran `mpi` module or `mpif.h`, here are examples of how each of the user-defined subroutines should be declared.

The user-function argument to MPI_OP_CREATE should be declared like this:

```
SUBROUTINE USER_FUNCTION(INVEC, INOUTVEC, LEN, DATATYPE)
   <type> INVEC(LEN), INOUTVEC(LEN)
   INTEGER LEN, DATATYPE
```

The copy and delete function arguments to MPI_COMM_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE COMM_COPY_ATTR_FUNCTION(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_FUNCTION(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_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE WIN_COPY_ATTR_FUNCTION(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_FUNCTION(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_CREATE_KEYVAL should be declared like these:

```
SUBROUTINE TYPE_COPY_ATTR_FUNCTION(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

SUBROUTINE TYPE_DELETE_ATTR_FUNCTION(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL,
              EXTRA_STATE, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

The handler-function argument to MPI_COMM_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE COMM_ERRHANDLER_FUNCTION(COMM, ERROR_CODE)
    INTEGER COMM, ERROR_CODE
```

The handler-function argument to MPI_WIN_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE WIN_ERRHANDLER_FUNCTION(WIN, ERROR_CODE)
    INTEGER WIN, ERROR_CODE
```

The handler-function argument to MPI_FILE_CREATE_ERRHANDLER should be declared like this:

```
SUBROUTINE FILE_ERRHANDLER_FUNCTION(FILE, ERROR_CODE)
    INTEGER FILE, ERROR_CODE
```

The query, free, and cancel function arguments to MPI_GREQUEST_START should be declared like these:

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

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

SUBROUTINE GREQUEST_CANCEL_FUNCTION(EXTRA_STATE, COMPLETE, IERROR)
   INTEGER IERROR
   INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
   LOGICAL COMPLETE
```

The extent and conversion function arguments to MPI_REGISTER_DATAREP should be declared like these:

```
SUBROUTINE DATAREP_EXTENT_FUNCTION(DATATYPE, EXTENT, EXTRA_STATE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT, EXTRA_STATE

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
```

### 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);
```

The following are deprecated Fortran user-defined callback subroutine prototypes. The deprecated copy and delete function arguments to MPI_KEYVAL_CREATE should be declared like these:

```
SUBROUTINE COPY_FUNCTION(OLDCOMM, KEYVAL, EXTRA_STATE,
               ATTRIBUTE_VAL_IN, ATTRIBUTE_VAL_OUT, FLAG, IERR)
   INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
         ATTRIBUTE_VAL_OUT, IERR
   LOGICAL FLAG

SUBROUTINE DELETE_FUNCTION(COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERR
```

### A.1.5 Info Keys

The following info keys are reserved. They are strings.

access_style
accumulate_ops
accumulate_ordering
alloc_shared_noncontig
appnum
arch
cb_block_size
cb_buffer_size
cb_nodes
chunked_item
chunked_size
chunked
collective_buffering
file_perm
filename
file
host
io_node_list
ip_address
ip_port
mpi_assert_allow_overtaking
mpi_assert_exact_length
mpi_assert_no_any_source
mpi_assert_no_any_tag
mpi_assert_strict_start_ordering
mpi_optimization_goal
mpi_reuse_count
nb_proc
no_locks
num_io_nodes
path
same_disp_unit
same_size
soft
striping_factor
striping_unit
wdir

### A.1.6 Info Values

The following info values are reserved. They are strings.

false
random
rar

raw
read_mostly
read_once
reverse_sequential
same_op
same_op_no_op
sequential
true
war
waw
write_mostly
write_once

## A.2 C Bindings

### A.2.1 Point-to-Point Communication C Bindings

**C binding**

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

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

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(const MPI_Status *status, MPI_Datatype datatype,
             int *count)

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

int MPI_Improbe(int source, int tag, MPI_Comm comm, int *flag,
             MPI_Message *message, MPI_Status *status)

int MPI_Imrecv(void *buf, int count, MPI_Datatype datatype,
             MPI_Message *message, 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(const void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm, MPI_Request *request)

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

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

int MPI_Mprobe(int source, int tag, MPI_Comm comm, MPI_Message *message,
             MPI_Status *status)

int MPI_Mrecv(void *buf, int count, MPI_Datatype datatype,
             MPI_Message *message, MPI_Status *status)

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, 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_Request_free(MPI_Request *request)

int MPI_Request_get_status(MPI_Request request, int *flag,
            MPI_Status *status)

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

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

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

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

int MPI_Sendrecv(const 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_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_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest,
            int tag, MPI_Comm comm)

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

int MPI_Start(MPI_Request *request)

int MPI_Startall(int count, MPI_Request array_of_requests[])

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

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

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_Testsome(int incount, MPI_Request array_of_requests[],
            int *outcount, int array_of_indices[],
            MPI_Status array_of_statuses[])

int MPI_Wait(MPI_Request *request, MPI_Status *status)

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_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

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)

MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)

int MPI_Get_address(const void *location, MPI_Aint *address)

int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outsize, int *position, MPI_Comm comm)

int MPI_Pack_external(const char *datarep, const void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf, MPI_Aint outsize,
             MPI_Aint *position)

int MPI_Pack_external_size(const char *datarep, int incount,
             MPI_Datatype datatype, MPI_Aint *size)

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,
             int *size)

int MPI_Type_commit(MPI_Datatype *datatype)

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

int MPI_Type_create_darray(int size, int rank, int ndims,
             const int array_of_gsizes[], const int array_of_distribs[],
             const int array_of_dargs[], const int array_of_psizes[],
             int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_create_hindexed(int count, const int array_of_blocklengths[],
             const MPI_Aint array_of_displacements[], MPI_Datatype oldtype,
             MPI_Datatype *newtype)

int MPI_Type_create_hindexed_block(int count, int blocklength,
             const 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,
             const 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, const int array_of_blocklengths[],
            const MPI_Aint array_of_displacements[],
            const MPI_Datatype array_of_types[], MPI_Datatype *newtype)

int MPI_Type_create_subarray(int ndims, const int array_of_sizes[],
            const int array_of_subsizes[], const int array_of_starts[],
            int order, MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Type_dup(MPI_Datatype oldtype, 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_elements(MPI_Status *status, MPI_Datatype datatype,
            int *count)

int MPI_Type_get_elements_x(MPI_Status *status, MPI_Datatype datatype,
            MPI_Count *count)

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_extent_x(MPI_Datatype datatype, MPI_Count *lb,
            MPI_Count *extent)

int MPI_Type_get_true_extent(MPI_Datatype datatype, MPI_Aint *true_lb,
            MPI_Aint *true_extent)

int MPI_Type_get_true_extent_x(MPI_Datatype datatype, MPI_Count *true_lb,
            MPI_Count *true_extent)

int MPI_Type_indexed(int count, const int array_of_blocklengths[],
            const int array_of_displacements[], MPI_Datatype oldtype,
            MPI_Datatype *newtype)

int MPI_Type_size(MPI_Datatype datatype, int *size)

int MPI_Type_size_x(MPI_Datatype datatype, MPI_Count *size)

int MPI_Type_vector(int count, int blocklength, int stride,
            MPI_Datatype oldtype, MPI_Datatype *newtype)

int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,
            int outcount, MPI_Datatype datatype, MPI_Comm comm)

int MPI_Unpack_external(const char *datarep, const void *inbuf,
            MPI_Aint insize, MPI_Aint *position, void *outbuf,
            int outsize, MPI_Datatype datatype)
```

### A.2.3 Collective Communication C Bindings

```
int MPI_Allgather(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Allgather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)

int MPI_Allgatherv(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Allgatherv_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Info info, MPI_Request* request)

int MPI_Allreduce(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)

int MPI_Alltoall(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
             void* recvbuf, int recvcount, MPI_Datatype recvtype,
             MPI_Comm comm)

int MPI_Alltoall_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)

int MPI_Alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
             const int recvcounts[], const int rdispls[],
             MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Alltoallv_init(const void* sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
             const int recvcounts[], const int rdispls[],
             MPI_Datatype recvtype, MPI_Comm comm, MPI_info info,
             MPI_Request *request)

int MPI_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[], MPI_Comm comm)

int MPI_Alltoallw_init(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[], MPI_Comm comm, MPI_Info info,
            MPI_Request *request)

int MPI_Barrier(MPI_Comm comm)

int MPI_Barrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
            MPI_Comm comm)

int MPI_Bcast_init(void* buffer, int count, MPI_Datatype datatype,
            int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)

int MPI_Exscan(const void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Exscan_init(const void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
            MPI_Info info, MPI_Request *request)

int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
            MPI_Comm comm)

int MPI_Gather_init(const void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, int recvcount,
            MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
            MPI_Request *request)

int MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, const int recvcounts[], const int displs[],
            MPI_Datatype recvtype, int root, MPI_Comm comm)

int MPI_Gatherv_init(const void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
            const int displs[], MPI_Datatype recvtype, int root,
            MPI_Comm comm, MPI_Info info, MPI_Request *request)

int MPI_Iallgather(const void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, int recvcount,
            MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

int MPI_Iallgatherv(const void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
            const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
            MPI_Request* request)

int MPI_Iallreduce(const void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
            MPI_Request *request)

int MPI_Ialltoall(const void* sendbuf, int sendcount,
```

```
              MPI_Datatype sendtype, void* recvbuf, int recvcount,
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

int MPI_Ialltoallv(const void* sendbuf, const int sendcounts[],
              const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
              const int recvcounts[], const int rdispls[],
              MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

int MPI_Ialltoallw(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[], MPI_Comm comm,
              MPI_Request *request)

int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)

int MPI_Ibcast(void* buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm, MPI_Request *request)

int MPI_Iexscan(const void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
              MPI_Request *request)

int MPI_Igather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm, MPI_Request *request)

int MPI_Igatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, const int recvcounts[], const int displs[],
              MPI_Datatype recvtype, int root, MPI_Comm comm,
              MPI_Request *request)

int MPI_Ireduce(const void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
              MPI_Request *request)

int MPI_Ireduce_scatter(const void* sendbuf, void* recvbuf,
              const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
              MPI_Comm comm, MPI_Request *request)

int MPI_Ireduce_scatter_block(const void* sendbuf, void* recvbuf,
              int recvcount, MPI_Datatype datatype, MPI_Op op,
              MPI_Comm comm, MPI_Request *request)

int MPI_Iscan(const void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
              MPI_Request *request)

int MPI_Iscatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm, MPI_Request *request)

int MPI_Iscatterv(const void* sendbuf, const int sendcounts[],
              const int displs[], MPI_Datatype sendtype, void* recvbuf,
```

```
            int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
            MPI_Request *request)

int MPI_Op_commutative(MPI_Op op, int *commute)

int MPI_Op_create(MPI_User_function* user_fn, int commute, MPI_Op* op)

int MPI_Op_free(MPI_Op *op)

int MPI_Reduce(const void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

int MPI_Reduce_init(const void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
            MPI_Info info, MPI_Request *request)

int MPI_Reduce_local(const void* inbuf, void* inoutbuf, int count,
            MPI_Datatype datatype, MPI_Op op)

int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,
            const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
            MPI_Comm comm)

int MPI_Reduce_scatter_block(const void* sendbuf, void* recvbuf,
            int recvcount, MPI_Datatype datatype, MPI_Op op,
            MPI_Comm comm)

int MPI_Reduce_scatter_block_init(const void* sendbuf, void* recvbuf,
            int recvcount, MPI_Datatype datatype, MPI_Op op,
            MPI_Comm comm, MPI_Info info, MPI_Request *request)

int MPI_Reduce_scatter_init(const void* sendbuf, void* recvbuf,
            const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
            MPI_Comm comm, MPI_Info info, MPI_Request *request)

int MPI_Scan(const void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

int MPI_Scan_init(const void* sendbuf, void* recvbuf, int count,
            MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
            MPI_Info info, MPI_Request *request)

int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
            void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
            MPI_Comm comm)

int MPI_Scatter_init(const void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, int recvcount,
            MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
            MPI_Request *request)

int MPI_Scatterv(const void* sendbuf, const int sendcounts[],
            const int displs[], MPI_Datatype sendtype, void* recvbuf,
            int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
int MPI_Scatterv_init(const void* sendbuf, const int sendcounts[],
              const int displs[], MPI_Datatype sendtype, void* recvbuf,
              int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
              MPI_Info info, MPI_Request *request)
```

### A.2.4 Groups, Contexts, Communicators, and Caching C Bindings

```
int MPI_COMM_DUP_FN(MPI_Comm oldcomm, int comm_keyval, void *extra_state,
              void *attribute_val_in, void *attribute_val_out, int *flag)

int MPI_COMM_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_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag,
              MPI_Comm *newcomm)

int MPI_Comm_create_keyval(MPI_Comm_copy_attr_function *comm_copy_attr_fn,
              MPI_Comm_delete_attr_function *comm_delete_attr_fn,
              int *comm_keyval, void *extra_state)

int MPI_Comm_delete_attr(MPI_Comm comm, int comm_keyval)

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

int MPI_Comm_dup_with_info(MPI_Comm comm, MPI_Info info, MPI_Comm *newcomm)

int MPI_Comm_free(MPI_Comm *comm)

int MPI_Comm_free_keyval(int *comm_keyval)

int MPI_Comm_get_attr(MPI_Comm comm, int comm_keyval, void* attribute_val,
              int *flag)

int MPI_Comm_get_info(MPI_Comm comm, MPI_Info *info_used)

int MPI_Comm_get_name(MPI_Comm comm, char *comm_name, int *resultlen)

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

int MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm, MPI_Request *request)

int MPI_Comm_idup_with_info(MPI_Comm comm, MPI_Info info,
              MPI_Comm *newcomm, MPI_Request *request)

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_info(MPI_Comm comm, MPI_Info info)

int MPI_Comm_set_name(MPI_Comm comm, const 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_split_type(MPI_Comm comm, int split_type, int key,
             MPI_Info info, 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, const int ranks[],
             MPI_Group *newgroup)

int MPI_Group_free(MPI_Group *group)

int MPI_Group_incl(MPI_Group group, int n, const 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, const 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 *newintracomm)

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_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_NULL_DELETE_FN(MPI_Datatype datatype, int type_keyval,
             void *attribute_val, void *extra_state)

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 datatype, int type_keyval)

int MPI_Type_free_keyval(int *type_keyval)

int MPI_Type_get_attr(MPI_Datatype datatype, int type_keyval,
             void* attribute_val, int *flag)

int MPI_Type_get_name(MPI_Datatype datatype, char *type_name,
             int *resultlen)

int MPI_Type_set_attr(MPI_Datatype datatype, int type_keyval,
             void* attribute_val)

int MPI_Type_set_name(MPI_Datatype datatype, const char *type_name)

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_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_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_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_set_attr(MPI_Win win, int win_keyval, void* attribute_val)

int MPI_Win_set_name(MPI_Win win, const 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, const int ndims, const int dims[],
            int periods[], int reorder, MPI_Comm *comm_cart)

int MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[],
            int coords[])

int MPI_Cart_map(MPI_Comm comm, int ndims, const int dims[],
            const int periods[], int *newrank)

int MPI_Cart_rank(MPI_Comm comm, const 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, const int remain_dims[], MPI_Comm *newcomm)

int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

int MPI_Dims_create(int nnodes, int ndims, int dims[])

int MPI_Dist_graph_create(MPI_Comm comm_old, int n, const int sources[],
            const int degrees[], const int destinations[],
            const int weights[], MPI_Info info, int reorder,
            MPI_Comm *comm_dist_graph)

int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old, int indegree,
            const int sources[], const int sourceweights[], int outdegree,
            const int destinations[], const int destweights[],
            MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)

int MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int sources[],
            int sourceweights[], int maxoutdegree, int destinations[],
            int destweights[])

int MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree,
            int *outdegree, int *weighted)

int MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int index[],
            const int edges[], int reorder, MPI_Comm *comm_graph)

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int index[],
            int edges[])

int MPI_Graph_map(MPI_Comm comm, int nnodes, const int index[],
            const int edges[], int *newrank)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
            int neighbors[])

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

int MPI_Ineighbor_allgather(const void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, int recvcount,
            MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
```

```
int MPI_Ineighbor_allgatherv(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Request *request)

int MPI_Ineighbor_alltoall(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

int MPI_Ineighbor_alltoallv(const void* sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
             const int recvcounts[], const int rdispls[],
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)

int MPI_Ineighbor_alltoallw(const void* sendbuf, const int sendcounts[],
             const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
             void* recvbuf, const int recvcounts[],
             const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
             MPI_Comm comm, MPI_Request *request)

int MPI_Neighbor_allgather(const void *sendbuf, int sendcount,
             MPI_Datatype sendtype, void *recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Neighbor_allgather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)

int MPI_Neighbor_allgatherv(const void *sendbuf, int sendcount,
             MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Neighbor_allgatherv_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)

int MPI_Neighbor_alltoall(const void *sendbuf, int sendcount,
             MPI_Datatype sendtype, void *recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Neighbor_alltoall_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)

int MPI_Neighbor_alltoallv(const void *sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void *recvbuf,
             const int recvcounts[], const int rdispls[],
             MPI_Datatype recvtype, MPI_Comm comm)

int MPI_Neighbor_alltoallv_init(const void* sendbuf,
```

```
            const int sendcounts[], const int sdispls[],
            MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
            const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
            MPI_Info info, MPI_Request *request)

int MPI_Neighbor_alltoallw(const void* sendbuf, const int sendcounts[],
            const MPI_Aint sdispls[], const MPI_Datatype sendtypes[],
            void* recvbuf, const int recvcounts[],
            const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
            MPI_Comm comm)

int MPI_Neighbor_alltoallw_init(const void* sendbuf,
            const int sendcounts[], const MPI_Aint sdispls[],
            const MPI_Datatype sendtypes[], void* recvbuf,
            const int recvcounts[], const MPI_Aint rdispls[],
            const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
            MPI_Request *request)

int MPI_Topo_test(MPI_Comm comm, int *status)
```

### A.2.6 MPI Environmental Management C Bindings

```
double MPI_Wtick(void)

double MPI_Wtime(void)

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, const 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_function *comm_errhandler_fn,
            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_function *file_errhandler_fn,
```

```
            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_Finalize(void)

int MPI_Finalized(int *flag)

int MPI_Free_mem(void *base)

int MPI_Get_library_version(char *version, int *resultlen)

int MPI_Get_processor_name(char *name, int *resultlen)

int MPI_Get_version(int *version, int *subversion)

int MPI_Init(int *argc, char ***argv)

int MPI_Initialized(int *flag)

int MPI_Win_call_errhandler(MPI_Win win, int errorcode)

int
            MPI_Win_create_errhandler(MPI_Win_errhandler_function *win_errhandler_fn,
            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)
```

### A.2.7 The Info Object C Bindings

```
int MPI_Info_create(MPI_Info *info)

int MPI_Info_delete(MPI_Info info, const 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, const 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, const char *key, int *valuelen,
            int *flag)

int MPI_Info_set(MPI_Info info, const char *key, const char *value)
```

### A.2.8 Process Creation and Management C Bindings

```
int MPI_Close_port(const char *port_name)
```

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

int MPI_Comm_connect(const 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_join(int fd, MPI_Comm *intercomm)

```
int MPI_Comm_spawn(const 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[], const int array_of_maxprocs[],
             const MPI_Info array_of_info[], int root, MPI_Comm comm,
             MPI_Comm *intercomm, int array_of_errcodes[])

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

int MPI_Open_port(MPI_Info info, char *port_name)

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

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

### A.2.9 One-Sided Communications C Bindings

```
int MPI_Accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

int MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
             void *result_addr, MPI_Datatype datatype, int target_rank,
             MPI_Aint target_disp, MPI_Win win)

int MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
             MPI_Datatype datatype, int target_rank, MPI_Aint target_disp,
             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_Get_accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, void *result_addr,
             int result_count, MPI_Datatype result_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)

int MPI_Put(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win)

int MPI_Raccumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
             MPI_Request *request)

int MPI_Rget(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_Request *request)

int MPI_Rget_accumulate(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, void *result_addr,
             int result_count, MPI_Datatype result_datatype,
             int target_rank, MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Op op, MPI_Win win,
             MPI_Request *request)

int MPI_Rput(const void *origin_addr, int origin_count,
             MPI_Datatype origin_datatype, int target_rank,
             MPI_Aint target_disp, int target_count,
             MPI_Datatype target_datatype, MPI_Win win,
             MPI_Request *request)

int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, void *baseptr, MPI_Win *win)

int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
             MPI_Comm comm, void *baseptr, MPI_Win *win)

int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)

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_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)

int MPI_Win_detach(MPI_Win win, const void *base)

int MPI_Win_fence(int assert, MPI_Win win)
```

```
int MPI_Win_flush(int rank, MPI_Win win)

int MPI_Win_flush_all(MPI_Win win)

int MPI_Win_flush_local(int rank, MPI_Win win)

int MPI_Win_flush_local_all(MPI_Win win)

int MPI_Win_free(MPI_Win *win)

int MPI_Win_get_group(MPI_Win win, MPI_Group *group)

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

int MPI_Win_lock_all(int assert, MPI_Win win)

int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_set_info(MPI_Win win, MPI_Info info)

int MPI_Win_shared_query(MPI_Win win, int rank, MPI_Aint *size,
            int *disp_unit, void *baseptr)

int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)

int MPI_Win_sync(MPI_Win win)

int MPI_Win_test(MPI_Win win, int *flag)

int MPI_Win_unlock(int rank, MPI_Win win)

int MPI_Win_unlock_all(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)

int MPI_Status_set_elements_x(MPI_Status *status, MPI_Datatype datatype,
            MPI_Count count)
```

### A.2.11 I/O C Bindings

```
int MPI_CONVERSION_FN_NULL(void *userbuf, MPI_Datatype datatype, int count,
              void *filebuf, MPI_Offset position, void *extra_state)

int MPI_File_close(MPI_File *fh)

int MPI_File_delete(const char *filename, MPI_Info info)

int MPI_File_get_amode(MPI_File fh, int *amode)

int MPI_File_get_atomicity(MPI_File fh, int *flag)

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

int MPI_File_get_group(MPI_File fh, MPI_Group *group)

int MPI_File_get_info(MPI_File fh, MPI_Info *info_used)

int MPI_File_get_position(MPI_File fh, MPI_Offset *offset)

int MPI_File_get_position_shared(MPI_File fh, MPI_Offset *offset)

int MPI_File_get_size(MPI_File fh, MPI_Offset *size)

int MPI_File_get_type_extent(MPI_File fh, MPI_Datatype datatype,
              MPI_Aint *extent)

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

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

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

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_at_all(MPI_File fh, MPI_Offset offset, 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(MPI_File fh, const void *buf, int count,
              MPI_Datatype datatype, MPI_Request *request)

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

int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, const void *buf,
              int count, MPI_Datatype datatype, MPI_Request *request)

int MPI_File_iwrite_at_all(MPI_File fh, MPI_Offset offset, const void *buf,
              int count, MPI_Datatype datatype, MPI_Request *request)
```

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

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

int MPI_File_preallocate(MPI_File fh, MPI_Offset size)

int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
             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_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_at(MPI_File fh, MPI_Offset offset, void *buf, int count,
             MPI_Datatype datatype, 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_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_ordered(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_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, const char *datarep, MPI_Info info)

int MPI_File_sync(MPI_File fh)

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

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

int MPI_File_write_all_begin(MPI_File fh, const void *buf, int count,
             MPI_Datatype datatype)

int MPI_File_write_all_end(MPI_File fh, const void *buf,
             MPI_Status *status)

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

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

int MPI_File_write_at_all_begin(MPI_File fh, MPI_Offset offset,
             const void *buf, int count, MPI_Datatype datatype)

int MPI_File_write_at_all_end(MPI_File fh, const void *buf,
             MPI_Status *status)

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

int MPI_File_write_ordered_begin(MPI_File fh, const void *buf, int count,
             MPI_Datatype datatype)

int MPI_File_write_ordered_end(MPI_File fh, const void *buf,
             MPI_Status *status)

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

int MPI_Register_datarep(const char *datarep,
             MPI_Datarep_conversion_function *read_conversion_fn,
             MPI_Datarep_conversion_function *write_conversion_fn,
             MPI_Datarep_extent_function *dtype_file_extent_fn,
             void *extra_state)
```

### A.2.12 Language Bindings C Bindings

```
int MPI_Status_f082f(MPI_F08_status *f08_status, MPI_Fint *f_status)

int MPI_Status_f2f08(MPI_Fint *f_status, MPI_F08_status *f08_status)

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 *datatype)

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_Message_c2f(MPI_Message message)

MPI_Message MPI_Message_f2c(MPI_Fint message)

MPI_Fint MPI_Op_c2f(MPI_Op op)

MPI_Op MPI_Op_f2c(MPI_Fint op)

MPI_Fint MPI_Request_c2f(MPI_Request request)

MPI_Request MPI_Request_f2c(MPI_Fint request)

int MPI_Status_c2f(const MPI_Status *c_status, MPI_Fint *f_status)

int MPI_Status_c2f08(const MPI_Status *c_status,
             MPI_F08_status *f08_status)

int MPI_Status_f082c(const MPI_F08_status *f08_status,
             MPI_Status *c_status)

int MPI_Status_f2c(const MPI_Fint *f_status, MPI_Status *c_status)

MPI_Fint MPI_Type_c2f(MPI_Datatype datatype)

MPI_Datatype MPI_Type_f2c(MPI_Fint datatype)

MPI_Fint MPI_Win_c2f(MPI_Win win)

MPI_Win MPI_Win_f2c(MPI_Fint win)
```

### A.2.13 Tools / Profiling Interface C Bindings

```
int MPI_Pcontrol(const int level, ...)
```

### A.2.14 Tools / MPI Tool Information Interface C Bindings

```
int MPI_T_category_changed(int *stamp)

int MPI_T_category_get_categories(int cat_index, int len, int indices[])

int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
```

```
int MPI_T_category_get_index(const char *name, int *cat_index)

int MPI_T_category_get_info(int cat_index, char *name, int *name_len,
             char *desc, int *desc_len, int *num_cvars, int *num_pvars,
             int *num_categories)

int MPI_T_category_get_num(int *num_cat)

int MPI_T_category_get_pvars(int cat_index, int len, int indices[])

int MPI_T_cvar_get_index(const char *name, int *cvar_index)

int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len,
             int *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype,
             char *desc, int *desc_len, int *bind, int *scope)

int MPI_T_cvar_get_num(int *num_cvar)

int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
             MPI_T_cvar_handle *handle, int *count)

int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)

int MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf)

int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf)

int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name,
             int *name_len)

int MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value,
             char *name, int *name_len)

int MPI_T_finalize(void)

int MPI_T_init_thread(int required, int *provided)

int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)

int MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len,
             int *verbosity, int *var_class, MPI_Datatype *datatype,
             MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind,
             int *readonly, int *continuous, int *atomic)

int MPI_T_pvar_get_num(int *num_pvar)

int MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index,
             void *obj_handle, MPI_T_pvar_handle *handle, int *count)

int MPI_T_pvar_handle_free(MPI_T_pvar_session session,
             MPI_T_pvar_handle *handle)

int MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle,
             void* buf)

int MPI_T_pvar_readreset(MPI_T_pvar_session session,
             MPI_T_pvar_handle handle, void* buf)
```

```
int MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)

int MPI_T_pvar_session_create(MPI_T_pvar_session *session)

int MPI_T_pvar_session_free(MPI_T_pvar_session *session)

int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)

int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)

int MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,
             const void* buf)
```

### A.2.15 Deprecated C Bindings

```
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_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)
```

## A.3 Fortran 2008 Bindings with the mpi_f08 Module

### A.3.1 Point-to-Point Communication Fortran 2008 Bindings

**F08 binding**

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

MPI_Bsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Buffer_attach(buffer, size, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Buffer_detach(buffer_addr, size, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    TYPE(C_PTR), INTENT(OUT) :: buffer_addr
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cancel(request, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_count(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) :: status
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(OUT) :: count
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Improbe(source, tag, comm, flag, message, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Message), INTENT(OUT) :: message
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Imrecv(buf, count, datatype, message, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Message), INTENT(INOUT) :: message
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Iprobe(source, tag, comm, flag, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Irsend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

```
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Mprobe(source, tag, comm, message, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Message), INTENT(OUT) :: message
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Mrecv(buf, count, datatype, message, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Message), INTENT(INOUT) :: message
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Probe(source, tag, comm, status, ierror)
    INTEGER, INTENT(IN) :: source, tag
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Recv_init(buf, count, datatype, source, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, source, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Request_free(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Request_get_status(request, flag, status, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Rsend(buf, count, datatype, dest, tag, comm, ierror)
```

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

MPI_Rsend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

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

MPI_Send_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcount, dest, sendtag, recvcount, source,
    recvtag
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag, source, recvtag,
             comm, status, ierror)
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count, dest, sendtag, source, recvtag
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Ssend(buf, count, datatype, dest, tag, comm, ierror)
```

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

MPI_Ssend_init(buf, count, datatype, dest, tag, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count, dest, tag
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Start(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Startall(count, array_of_requests, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Test(request, flag, status, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

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

MPI_Testall(count, array_of_requests, flag, array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Testany(count, array_of_requests, index, flag, status, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: index
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Testsome(incount, array_of_requests, outcount, array_of_indices,
             array_of_statuses, ierror)
```

```
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Wait(request, status, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitany(count, array_of_requests, index, status, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: index
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices,
             array_of_statuses, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_requests(count)
    INTEGER, INTENT(OUT) :: outcount, array_of_indices(*)
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### A.3.2 Datatypes Fortran 2008 Bindings

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_add(base, disp)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: base, disp

INTEGER(KIND=MPI_ADDRESS_KIND) MPI_Aint_diff(addr1, addr2)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: addr1, addr2

MPI_Get_address(location, address, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: location
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: address
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
    INTEGER, INTENT(IN) :: incount, outsize
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(*), DIMENSION(..) :: outbuf
    INTEGER, INTENT(INOUT) :: position
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack_external(datarep, inbuf, incount, datatype, outbuf, outsize,
             position, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    TYPE(*), DIMENSION(..), INTENT(IN) :: inbuf
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(*), DIMENSION(..) :: outbuf
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: outsize
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) :: position
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack_external_size(datarep, incount, datatype, size, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: datarep
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Pack_size(incount, datatype, comm, size, ierror)
    INTEGER, INTENT(IN) :: incount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_commit(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_contiguous(count, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_darray(size, rank, ndims, array_of_gsizes,
             array_of_distribs, array_of_dargs, array_of_psizes, order,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: size, rank, ndims, array_of_gsizes(ndims),
    array_of_distribs(ndims), array_of_dargs(ndims),
    array_of_psizes(ndims), order
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_hindexed(count, array_of_blocklengths,
             array_of_displacements, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_hindexed_block(count, blocklength, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count, blocklength
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_hvector(count, blocklength, stride, oldtype, newtype,
             ierror)
    INTEGER, INTENT(IN) :: count, blocklength
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: stride
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_indexed_block(count, blocklength, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: count, blocklength,
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_resized(oldtype, lb, extent, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: oldtype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: lb, extent
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_struct(count, array_of_blocklengths,
             array_of_displacements, array_of_types, newtype, ierror)
    INTEGER, INTENT(IN) :: count, array_of_blocklengths(count)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) :: array_of_types(count)
    TYPE(MPI_Datatype), INTENT(OUT) :: newtype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_create_subarray(ndims, array_of_sizes, array_of_subsizes,
             array_of_starts, order, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) :: ndims, array_of_sizes(ndims),
    array_of_subsizes(ndims), array_of_starts(ndims), order
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_dup(oldtype, newtype, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_free(datatype, ierror)
    TYPE(MPI_Datatype), INTENT(INOUT) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_contents(datatype, max_integers, max_addresses, max_datatypes,
             array_of_integers, array_of_addresses, array_of_datatypes,
             ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  max_integers, max_addresses, max_datatypes
    INTEGER, INTENT(OUT) ::  array_of_integers(max_integers)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::
    array_of_addresses(max_addresses)
    TYPE(MPI_Datatype), INTENT(OUT) ::  array_of_datatypes(max_datatypes)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_elements(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_elements_x(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_envelope(datatype, num_integers, num_addresses, num_datatypes,
             combiner, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  num_integers, num_addresses, num_datatypes,
    combiner
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_extent(datatype, lb, extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  lb, extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_extent_x(datatype, lb, extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::  lb, extent
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_true_extent(datatype, true_lb, true_extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  true_lb, true_extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_get_true_extent_x(datatype, true_lb, true_extent, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::  true_lb, true_extent
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements,
             oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, array_of_blocklengths(count),
    array_of_displacements(count)
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_size(datatype, size, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_size_x(datatype, size, ierror)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_vector(count, blocklength, stride, oldtype, newtype, ierror)
    INTEGER, INTENT(IN) ::  count, blocklength, stride
    TYPE(MPI_Datatype), INTENT(IN) ::  oldtype
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype, comm,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    INTEGER, INTENT(IN) ::  insize, outcount
    INTEGER, INTENT(INOUT) ::  position
    TYPE(*), DIMENSION(..)  ::  outbuf
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Unpack_external(datarep, inbuf, insize, position, outbuf, outsize,
             datatype, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  datarep
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  insize
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(INOUT) ::  position
    TYPE(*), DIMENSION(..)  ::  outbuf
    INTEGER, INTENT(IN) ::  outsize
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

### A.3.3 Collective Communication Fortran 2008 Bindings

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Allgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Allgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
             displs, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
             rdispls, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), sdispls(*), recvcounts(*),
    rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Alltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
             recvcounts, rdispls, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts,
             rdispls, recvtypes, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcounts(*), sdispls(*), recvcounts(*),
              rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtypes(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS ::  sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Barrier(comm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Barrier_init(comm, info, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Bcast(buffer, count, datatype, root, comm, ierror)
    TYPE(*), DIMENSION(..)  ::  buffer
```

```
    INTEGER, INTENT(IN) ::  count, root
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Bcast_init(buffer, count, datatype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buffer
    INTEGER, INTENT(IN) ::  count, root
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Exscan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Gather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Gatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcounts(*), displs(*), root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Gatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Iallgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Iallgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Iallreduce(sendbuf, recvbuf, count, datatype, op, comm, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ialltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ialltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts,
             rdispls, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ialltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ibarrier(comm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ibcast(buffer, count, datatype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
```

```
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Iexscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Igather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Igatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
             recvtype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Ireduce(sendbuf, recvbuf, count, datatype, op, root, comm, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  count, root
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Ireduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
```

```
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ireduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Iscan(sendbuf, recvbuf, count, datatype, op, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Iscatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Iscatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
             recvtype, root, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
    INTEGER, INTENT(IN) :: recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Op_commutative(op, commute, ierror)
```

```
    TYPE(MPI_Op), INTENT(IN) ::  op
    LOGICAL, INTENT(OUT) ::  commute
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Op_create(user_fn, commute, op, ierror)
    PROCEDURE(MPI_User_function) ::  user_fn
    LOGICAL, INTENT(IN) ::  commute
    TYPE(MPI_Op), INTENT(OUT) ::  op
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Op_free(op, ierror)
    TYPE(MPI_Op), INTENT(INOUT) ::  op
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  count, root
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  count, root
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Reduce_local(inbuf, inoutbuf, count, datatype, op, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  inbuf
    TYPE(*), DIMENSION(..)  ::  inoutbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Reduce_scatter_block(sendbuf, recvbuf, recvcount, datatype, op, comm,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
             comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
             recvtype, root, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    TYPE(*), DIMENSION(..) :: recvbuf
    INTEGER, INTENT(IN) :: sendcounts(*), displs(*), recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Scatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
             recvcount, recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
    INTEGER, INTENT(IN) :: recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### A.3.4 Groups, Contexts, Communicators, and Caching Fortran 2008 Bindings

```
MPI_COMM_DUP_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Comm) ::  oldcomm
    INTEGER ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val_out
    LOGICAL ::  flag
    INTEGER ::  ierror
```

```
MPI_COMM_NULL_COPY_FN(oldcomm, comm_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Comm) ::  oldcomm
    INTEGER ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val_out
    LOGICAL ::  flag
    INTEGER ::  ierror
```

```
MPI_COMM_NULL_DELETE_FN(comm, comm_keyval, attribute_val, extra_state,
             ierror)
    TYPE(MPI_Comm) ::  comm
    INTEGER ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
    INTEGER ::  ierror
```

```
MPI_Comm_compare(comm1, comm2, result, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm1, comm2
    INTEGER, INTENT(OUT) ::  result
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Comm_create(comm, group, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(IN) ::  group
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Comm_create_group(comm, group, tag, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  tag
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Comm_create_keyval(comm_copy_attr_fn, comm_delete_attr_fn, comm_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Comm_copy_attr_function) ::  comm_copy_attr_fn
    PROCEDURE(MPI_Comm_delete_attr_function) ::  comm_delete_attr_fn
    INTEGER, INTENT(OUT) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_delete_attr(comm, comm_keyval, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_dup(comm, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_dup_with_info(comm, info, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_free(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_free_keyval(comm_keyval, ierror)
    INTEGER, INTENT(INOUT) :: comm_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_get_attr(comm, comm_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_get_info(comm, info_used, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(OUT) :: info_used
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_get_name(comm, comm_name, resultlen, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: comm_name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_idup(comm, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
```

```
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_idup_with_info(comm, info, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS ::  newcomm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_rank(comm, rank, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  rank
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_remote_group(comm, group, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Group), INTENT(OUT) ::  group
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_remote_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_set_attr(comm, comm_keyval, attribute_val, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  comm_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_set_info(comm, info, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_set_name(comm, comm_name, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    CHARACTER(LEN=*), INTENT(IN) ::  comm_name
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_size(comm, size, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_split(comm, color, key, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  color, key
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Comm_split_type(comm, split_type, key, info, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  split_type, key
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Comm_test_inter(comm, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_compare(group1, group2, result, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    INTEGER, INTENT(OUT) ::  result
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_difference(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_excl(group, n, ranks, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranks(n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_free(group, ierror)
    TYPE(MPI_Group), INTENT(INOUT) ::  group
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_incl(group, n, ranks, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranks(n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_intersection(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group1, group2
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_range_excl(group, n, ranges, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranges(3,n)
    TYPE(MPI_Group), INTENT(OUT) ::  newgroup
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Group_range_incl(group, n, ranges, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  n, ranges(3,n)
```

```
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_rank(group, rank, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_size(group, size, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group
    INTEGER, INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_translate_ranks(group1, n, ranks1, group2, ranks2, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    INTEGER, INTENT(IN) :: n, ranks1(n)
    INTEGER, INTENT(OUT) :: ranks2(n)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Group_union(group1, group2, newgroup, ierror)
    TYPE(MPI_Group), INTENT(IN) :: group1, group2
    TYPE(MPI_Group), INTENT(OUT) :: newgroup
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Intercomm_create(local_comm, local_leader, peer_comm, remote_leader,
             tag, newintercomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: local_comm, peer_comm
    INTEGER, INTENT(IN) :: local_leader, remote_leader, tag
    TYPE(MPI_Comm), INTENT(OUT) :: newintercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Intercomm_merge(intercomm, high, newintracomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: intercomm
    LOGICAL, INTENT(IN) :: high
    TYPE(MPI_Comm), INTENT(OUT) :: newintracomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_TYPE_DUP_FN(oldtype, type_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Datatype) :: oldtype
    INTEGER :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
    LOGICAL :: flag
    INTEGER :: ierror

MPI_TYPE_NULL_COPY_FN(oldtype, type_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Datatype) :: oldtype
    INTEGER :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: extra_state, attribute_val_in
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val_out
    LOGICAL :: flag
    INTEGER :: ierror

MPI_TYPE_NULL_DELETE_FN(datatype, type_keyval, attribute_val, extra_state,
             ierror)
    TYPE(MPI_Datatype) :: datatype
    INTEGER :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) :: attribute_val, extra_state
    INTEGER, INTENT(OUT) :: ierror

MPI_Type_create_keyval(type_copy_attr_fn, type_delete_attr_fn, type_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Type_copy_attr_function) :: type_copy_attr_fn
    PROCEDURE(MPI_Type_delete_attr_function) :: type_delete_attr_fn
    INTEGER, INTENT(OUT) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: extra_state
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_delete_attr(datatype, type_keyval, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_free_keyval(type_keyval, ierror)
    INTEGER, INTENT(INOUT) :: type_keyval
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_get_attr(datatype, type_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: attribute_val
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_get_name(datatype, type_name, resultlen, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) :: type_name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_set_attr(datatype, type_keyval, attribute_val, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: type_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Type_set_name(datatype, type_name, ierror)
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    CHARACTER(LEN=*), INTENT(IN) :: type_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_WIN_DUP_FN(oldwin, win_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Win) ::  oldwin
    INTEGER ::  win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val_out
    LOGICAL ::  flag
    INTEGER ::  ierror

MPI_WIN_NULL_COPY_FN(oldwin, win_keyval, extra_state, attribute_val_in,
             attribute_val_out, flag, ierror)
    TYPE(MPI_Win) ::  oldwin
    INTEGER ::  win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state, attribute_val_in
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val_out
    LOGICAL ::  flag
    INTEGER ::  ierror

MPI_WIN_NULL_DELETE_FN(win, win_keyval, attribute_val, extra_state, ierror)
    TYPE(MPI_Win) ::  win
    INTEGER ::  win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  attribute_val, extra_state
    INTEGER ::  ierror

MPI_Win_create_keyval(win_copy_attr_fn, win_delete_attr_fn, win_keyval,
             extra_state, ierror)
    PROCEDURE(MPI_Win_copy_attr_function) ::  win_copy_attr_fn
    PROCEDURE(MPI_Win_delete_attr_function) ::  win_delete_attr_fn
    INTEGER, INTENT(OUT) ::  win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_delete_attr(win, win_keyval, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, INTENT(IN) ::  win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_free_keyval(win_keyval, ierror)
    INTEGER, INTENT(INOUT) ::  win_keyval
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_get_attr(win, win_keyval, attribute_val, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, INTENT(IN) ::  win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  attribute_val
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_get_name(win, win_name, resultlen, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    CHARACTER(LEN=MPI_MAX_OBJECT_NAME), INTENT(OUT) ::  win_name
```

```
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_attr(win, win_keyval, attribute_val, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: win_keyval
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: attribute_val
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_name(win, win_name, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    CHARACTER(LEN=*), INTENT(IN) :: win_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### A.3.5 Process Topologies Fortran 2008 Bindings

```
MPI_Cart_coords(comm, rank, maxdims, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: rank, maxdims
    INTEGER, INTENT(OUT) :: coords(maxdims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm_old
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
    LOGICAL, INTENT(IN) :: periods(ndims), reorder
    TYPE(MPI_Comm), INTENT(OUT) :: comm_cart
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_get(comm, maxdims, dims, periods, coords, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: maxdims
    INTEGER, INTENT(OUT) :: dims(maxdims), coords(maxdims)
    LOGICAL, INTENT(OUT) :: periods(maxdims)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_map(comm, ndims, dims, periods, newrank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: ndims, dims(ndims)
    LOGICAL, INTENT(IN) :: periods(ndims)
    INTEGER, INTENT(OUT) :: newrank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_rank(comm, coords, rank, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: coords(*)
    INTEGER, INTENT(OUT) :: rank
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Cart_shift(comm, direction, disp, rank_source, rank_dest, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
```

```
    INTEGER, INTENT(IN) ::  direction, disp
    INTEGER, INTENT(OUT) ::  rank_source, rank_dest
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Cart_sub(comm, remain_dims, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    LOGICAL, INTENT(IN) ::  remain_dims(*)
    TYPE(MPI_Comm), INTENT(OUT) ::  newcomm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Cartdim_get(comm, ndims, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  ndims
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dims_create(nnodes, ndims, dims, ierror)
    INTEGER, INTENT(IN) ::  nnodes, ndims
    INTEGER, INTENT(INOUT) ::  dims(ndims)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_create(comm_old, n, sources, degrees, destinations, weights,
             info, reorder, comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  n, sources(n), degrees(n), destinations(*),
    weights(*)
    TYPE(MPI_Info), INTENT(IN) ::  info
    LOGICAL, INTENT(IN) ::  reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_create_adjacent(comm_old, indegree, sources, sourceweights,
             outdegree, destinations, destweights, info, reorder,
             comm_dist_graph, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  indegree, sources(indegree), sourceweights(*),
    outdegree, destinations(outdegree), destweights(*)
    TYPE(MPI_Info), INTENT(IN) ::  info
    LOGICAL, INTENT(IN) ::  reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_dist_graph
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_neighbors(comm, maxindegree, sources, sourceweights,
             maxoutdegree, destinations, destweights, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  maxindegree, maxoutdegree
    INTEGER, INTENT(OUT) ::  sources(maxindegree),
    destinations(maxoutdegree)
    INTEGER ::  sourceweights(*), destweights(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Dist_graph_neighbors_count(comm, indegree, outdegree, weighted, ierror)
```

```
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  indegree, outdegree
    LOGICAL, INTENT(OUT) ::  weighted
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph,
             ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm_old
    INTEGER, INTENT(IN) ::  nnodes, index(nnodes), edges(*)
    LOGICAL, INTENT(IN) ::  reorder
    TYPE(MPI_Comm), INTENT(OUT) ::  comm_graph
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Graph_get(comm, maxindex, maxedges, index, edges, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  maxindex, maxedges
    INTEGER, INTENT(OUT) ::  index(maxindex), edges(maxedges)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Graph_map(comm, nnodes, index, edges, newrank, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  nnodes, index(nnodes), edges(*)
    INTEGER, INTENT(OUT) ::  newrank
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  rank, maxneighbors
    INTEGER, INTENT(OUT) ::  neighbors(maxneighbors)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Graph_neighbors_count(comm, rank, nneighbors, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(IN) ::  rank
    INTEGER, INTENT(OUT) ::  nneighbors
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Graphdims_get(comm, nnodes, nedges, ierror)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, INTENT(OUT) ::  nnodes, nedges
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Ineighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Ineighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
             displs, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ineighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ineighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
             recvcounts, rdispls, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Ineighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
              sdispls(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
              recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Neighbor_allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) :: sendbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
```

```
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_allgatherv(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
             displs, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcounts, displs, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
```

```
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_alltoallv(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
             recvcounts, rdispls, recvtype, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    INTEGER, INTENT(IN) ::  sendcounts(*), sdispls(*), recvcounts(*),
    rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(*), DIMENSION(..)  ::  recvbuf
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
             recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), sdispls(*),
              recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_alltoallw(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
             recvcounts, rdispls, recvtypes, comm, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN) ::  sendbuf
    TYPE(*), DIMENSION(..)  ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  sdispls(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtypes(*), recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Neighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
             recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS ::
              sdispls(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS ::  sendtypes(*),
```

```
            recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Topo_test(comm, status, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(OUT) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### A.3.6 MPI Environmental Management Fortran 2008 Bindings

```
DOUBLE PRECISION MPI_Wtick()

DOUBLE PRECISION MPI_Wtime()

MPI_Abort(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Add_error_class(errorclass, ierror)
    INTEGER, INTENT(OUT) :: errorclass
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Add_error_code(errorclass, errorcode, ierror)
    INTEGER, INTENT(IN) :: errorclass
    INTEGER, INTENT(OUT) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Add_error_string(errorcode, string, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=*), INTENT(IN) :: string
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Alloc_mem(size, info, baseptr, ierror)
    USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND, INTENT(IN) :: size
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(C_PTR), INTENT(OUT) :: baseptr
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_call_errhandler(comm, errorcode, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_create_errhandler(comm_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_Comm_errhandler_function) :: comm_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_Comm_get_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_set_errhandler(comm, errhandler, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Errhandler_free(errhandler, ierror)
    TYPE(MPI_Errhandler), INTENT(INOUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Error_class(errorcode, errorclass, ierror)
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, INTENT(OUT) :: errorclass
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Error_string(errorcode, string, resultlen, ierror)
    INTEGER, INTENT(IN) :: errorcode
    CHARACTER(LEN=MPI_MAX_ERROR_STRING), INTENT(OUT) :: string
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_call_errhandler(fh, errorcode, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_create_errhandler(file_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_File_errhandler_function) :: file_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_set_errhandler(file, errhandler, ierror)
    TYPE(MPI_File), INTENT(IN) :: file
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Finalize(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Finalized(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Free_mem(base, ierror)
```

```
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: base
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_library_version(version, resultlen, ierror)
    CHARACTER(LEN=MPI_MAX_LIBRARY_VERSION_STRING), INTENT(OUT) :: version
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_processor_name(name, resultlen, ierror)
    CHARACTER(LEN=MPI_MAX_PROCESSOR_NAME), INTENT(OUT) :: name
    INTEGER, INTENT(OUT) :: resultlen
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_version(version, subversion, ierror)
    INTEGER, INTENT(OUT) :: version, subversion
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Init(ierror)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Initialized(flag, ierror)
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_call_errhandler(win, errorcode, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, INTENT(IN) :: errorcode
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_create_errhandler(win_errhandler_fn, errhandler, ierror)
    PROCEDURE(MPI_Win_errhandler_function) :: win_errhandler_fn
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_get_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(OUT) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Win_set_errhandler(win, errhandler, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Errhandler), INTENT(IN) :: errhandler
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### A.3.7 The Info Object Fortran 2008 Bindings

```
MPI_Info_create(info, ierror)
    TYPE(MPI_Info), INTENT(OUT) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_delete(info, key, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
```

```
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_dup(info, newinfo, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Info), INTENT(OUT) :: newinfo
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_free(info, ierror)
    TYPE(MPI_Info), INTENT(INOUT) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get(info, key, valuelen, value, flag, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, INTENT(IN) :: valuelen
    CHARACTER(LEN=*), INTENT(OUT) :: value
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_nkeys(info, nkeys, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(OUT) :: nkeys
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_nthkey(info, n, key, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: n
    CHARACTER(LEN=*), INTENT(OUT) :: key
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_get_valuelen(info, key, valuelen, flag, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key
    INTEGER, INTENT(OUT) :: valuelen
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Info_set(info, key, value, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=*), INTENT(IN) :: key, value
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### A.3.8 Process Creation and Management Fortran 2008 Bindings

```
MPI_Close_port(port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_accept(port_name, info, root, comm, newcomm, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
```

```
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_connect(port_name, info, root, comm, newcomm, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: port_name
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, INTENT(IN) :: root
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_disconnect(comm, ierror)
    TYPE(MPI_Comm), INTENT(INOUT) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_get_parent(parent, ierror)
    TYPE(MPI_Comm), INTENT(OUT) :: parent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_join(fd, intercomm, ierror)
    INTEGER, INTENT(IN) :: fd
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_join(fd, intercomm, ierror)
    INTEGER, INTENT(IN) :: fd
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm,
             array_of_errcodes, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: command, argv(*)
    INTEGER, INTENT(IN) :: maxprocs, root
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
    INTEGER :: array_of_errcodes(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Comm_spawn_multiple(count, array_of_commands, array_of_argv,
             array_of_maxprocs, array_of_info, root, comm, intercomm,
             array_of_errcodes, ierror)
    INTEGER, INTENT(IN) :: count, array_of_maxprocs(*), root
    CHARACTER(LEN=*), INTENT(IN) :: array_of_commands(*),
    array_of_argv(count,*)
    TYPE(MPI_Info), INTENT(IN) :: array_of_info(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: intercomm
```

```
    INTEGER :: array_of_errcodes(*)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Lookup_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: service_name
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Open_port(info, port_name, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    CHARACTER(LEN=MPI_MAX_PORT_NAME), INTENT(OUT) :: port_name
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Publish_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Unpublish_name(service_name, info, port_name, ierror)
    CHARACTER(LEN=*), INTENT(IN) :: service_name, port_name
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

### A.3.9 One-Sided Communications Fortran 2008 Bindings

```
MPI_Accumulate(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Compare_and_swap(origin_addr, compare_addr, result_addr, datatype,
             target_rank, target_disp, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: compare_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: target_rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Fetch_and_op(origin_addr, result_addr, datatype, target_rank,
             target_disp, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, INTENT(IN) :: target_rank
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Get_accumulate(origin_addr, origin_count, origin_datatype, result_addr,
             result_count, result_datatype, target_rank, target_disp,
             target_count, target_datatype, op, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: result_addr
    INTEGER, INTENT(IN) :: origin_count, result_count, target_rank,
              target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype,
              result_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Put(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_Raccumulate(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: origin_addr
    INTEGER, INTENT(IN) :: origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) :: origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: target_disp
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Win), INTENT(IN) :: win
```

```
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Rget(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, request,
             ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Rget_accumulate(origin_addr, origin_count, origin_datatype,
             result_addr, result_count, result_datatype, target_rank,
             target_disp, target_count, target_datatype, op, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  result_addr
    INTEGER, INTENT(IN) ::  origin_count, result_count, target_rank,
              target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype,
              result_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Rput(origin_addr, origin_count, origin_datatype, target_rank,
             target_disp, target_count, target_datatype, win, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  origin_addr
    INTEGER, INTENT(IN) ::  origin_count, target_rank, target_count
    TYPE(MPI_Datatype), INTENT(IN) ::  origin_datatype, target_datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  target_disp
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_allocate(size, disp_unit, info, comm, baseptr, win, ierror)
    USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  size
    INTEGER, INTENT(IN) ::  disp_unit
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(C_PTR), INTENT(OUT) ::  baseptr
    TYPE(MPI_Win), INTENT(OUT) ::  win
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_allocate_shared(size, disp_unit, info, comm, baseptr, win, ierror)
    USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  size
    INTEGER, INTENT(IN) ::  disp_unit
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(C_PTR), INTENT(OUT) ::  baseptr
    TYPE(MPI_Win), INTENT(OUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_attach(win, base, size, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  base
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_complete(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_create(base, size, disp_unit, info, comm, win, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  base
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  size
    INTEGER, INTENT(IN) ::  disp_unit
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Win), INTENT(OUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_create_dynamic(info, comm, win, ierror)
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Win), INTENT(OUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_detach(win, base, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  base
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_fence(assert, win, ierror)
    INTEGER, INTENT(IN) ::  assert
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_flush(rank, win, ierror)
    INTEGER, INTENT(IN) ::  rank
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_Win_flush_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_flush_local(rank, win, ierror)
    INTEGER, INTENT(IN) ::  rank
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_flush_local_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_free(win, ierror)
    TYPE(MPI_Win), INTENT(INOUT) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_get_group(win, group, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Group), INTENT(OUT) ::  group
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_get_info(win, info_used, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Info), INTENT(OUT) ::  info_used
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_lock(lock_type, rank, assert, win, ierror)
    INTEGER, INTENT(IN) ::  lock_type, rank, assert
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_lock_all(assert, win, ierror)
    INTEGER, INTENT(IN) ::  assert
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_post(group, assert, win, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  assert
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_set_info(win, info, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_shared_query(win, rank, size, disp_unit, baseptr, ierror)
    USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, INTENT(IN) ::  rank
```

```
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) ::  size
    INTEGER, INTENT(OUT) ::  disp_unit
    TYPE(C_PTR), INTENT(OUT) ::  baseptr
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_start(group, assert, win, ierror)
    TYPE(MPI_Group), INTENT(IN) ::  group
    INTEGER, INTENT(IN) ::  assert
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_sync(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_test(win, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_unlock(rank, win, ierror)
    INTEGER, INTENT(IN) ::  rank
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_unlock_all(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Win_wait(win, ierror)
    TYPE(MPI_Win), INTENT(IN) ::  win
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

### A.3.10 External Interfaces Fortran 2008 Bindings

```
MPI_Grequest_complete(request, ierror)
    TYPE(MPI_Request), INTENT(IN) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Grequest_start(query_fn, free_fn, cancel_fn, extra_state, request,
             ierror)
    PROCEDURE(MPI_Grequest_query_function) ::  query_fn
    PROCEDURE(MPI_Grequest_free_function) ::  free_fn
    PROCEDURE(MPI_Grequest_cancel_function) ::  cancel_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Init_thread(required, provided, ierror)
    INTEGER, INTENT(IN) ::  required
    INTEGER, INTENT(OUT) ::  provided
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Is_thread_main(flag, ierror)
    LOGICAL, INTENT(OUT) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Query_thread(provided, ierror)
    INTEGER, INTENT(OUT) ::  provided
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Status_set_cancelled(status, flag, ierror)
    TYPE(MPI_Status), INTENT(INOUT) ::  status
    LOGICAL, INTENT(IN) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Status_set_elements(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(INOUT) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, INTENT(IN) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Status_set_elements_x(status, datatype, count, ierror)
    TYPE(MPI_Status), INTENT(INOUT) ::  status
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN) ::  count
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

### A.3.11 I/O Fortran 2008 Bindings

```
MPI_CONVERSION_FN_NULL(userbuf, datatype, count, filebuf, position,
             extra_state, ierror)
    USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
    TYPE(C_PTR), VALUE ::  userbuf, filebuf
    TYPE(MPI_Datatype) ::  datatype
    INTEGER ::  count, ierror
    INTEGER(KIND=MPI_OFFSET_KIND) ::  position
    INTEGER(KIND=MPI_ADDRESS_KIND) ::  extra_state

MPI_File_close(fh, ierror)
    TYPE(MPI_File), INTENT(INOUT) ::  fh
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_delete(filename, info, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  filename
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_get_amode(fh, amode, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER, INTENT(OUT) ::  amode
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_File_get_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_byte_offset(fh, offset, disp, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_group(fh, group, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Group), INTENT(OUT) :: group
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_info(fh, info_used, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Info), INTENT(OUT) :: info_used
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_position(fh, offset, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_position_shared(fh, offset, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: offset
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_size(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_type_extent(fh, datatype, extent, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(OUT) :: extent
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_get_view(fh, disp, etype, filetype, datarep, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(OUT) :: disp
    TYPE(MPI_Datatype), INTENT(OUT) :: etype, filetype
    CHARACTER(LEN=*), INTENT(OUT) :: datarep
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iread(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
```

```
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iread_all(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iread_at(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iread_at_all(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iread_shared(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite_all(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
```

```
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite_at(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite_at_all(fh, offset, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_iwrite_shared(fh, buf, count, datatype, request, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_open(comm, filename, amode, info, fh, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    CHARACTER(LEN=*), INTENT(IN) :: filename
    INTEGER, INTENT(IN) :: amode
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_File), INTENT(OUT) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_preallocate(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: size
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_all(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_all_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_at(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_at_all(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_at_all_begin(fh, offset, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_read_at_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_ordered(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_ordered_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_ordered_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_read_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..)  ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_seek(fh, offset, whence, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    INTEGER, INTENT(IN) ::  whence
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_seek_shared(fh, offset, whence, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  offset
    INTEGER, INTENT(IN) ::  whence
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_set_atomicity(fh, flag, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    LOGICAL, INTENT(IN) ::  flag
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_set_info(fh, info, ierror)
```

```
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_set_size(fh, size, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) ::  disp
    TYPE(MPI_Datatype), INTENT(IN) ::  etype, filetype
    CHARACTER(LEN=*), INTENT(IN) ::  datarep
    TYPE(MPI_Info), INTENT(IN) ::  info
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_sync(fh, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_write(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_write_all(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_write_all_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_write_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_write_at(fh, offset, buf, count, datatype, status, ierror)
```

```
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at_all(fh, offset, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at_all_begin(fh, offset, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER(KIND=MPI_OFFSET_KIND), INTENT(IN) :: offset
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_at_all_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_ordered(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN) :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_ordered_begin(fh, buf, count, datatype, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_File_write_ordered_end(fh, buf, status, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: buf
    TYPE(MPI_Status) :: status
```

```
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_File_write_shared(fh, buf, count, datatype, status, ierror)
    TYPE(MPI_File), INTENT(IN) ::  fh
    TYPE(*), DIMENSION(..), INTENT(IN) ::  buf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Register_datarep(datarep, read_conversion_fn, write_conversion_fn,
             dtype_file_extent_fn, extra_state, ierror)
    CHARACTER(LEN=*), INTENT(IN) ::  datarep
    PROCEDURE(MPI_Datarep_conversion_function) ::  read_conversion_fn
    PROCEDURE(MPI_Datarep_conversion_function) ::  write_conversion_fn
    PROCEDURE(MPI_Datarep_extent_function) ::  dtype_file_extent_fn
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) ::  extra_state
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

### A.3.12 Language Bindings Fortran 2008 Bindings

```
MPI_F_sync_reg(buf)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  buf

MPI_Sizeof(x, size, ierror)
    TYPE(*), DIMENSION(..)  ::  x
    INTEGER, INTENT(OUT) ::  size
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Status_f082f(f08_status, f_status, ierror)
    TYPE(MPI_Status), INTENT(IN) ::  f08_status
    INTEGER, INTENT(OUT) ::  f_status(MPI_STATUS_SIZE)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Status_f2f08(f_status, f08_status, ierror)
    INTEGER, INTENT(IN) ::  f_status(MPI_STATUS_SIZE)
    TYPE(MPI_Status), INTENT(OUT) ::  f08_status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_f90_complex(p, r, newtype, ierror)
    INTEGER, INTENT(IN) ::  p, r
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_f90_integer(r, newtype, ierror)
    INTEGER, INTENT(IN) ::  r
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_create_f90_real(p, r, newtype, ierror)
    INTEGER, INTENT(IN) ::  p, r
```

```
    TYPE(MPI_Datatype), INTENT(OUT) ::  newtype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_Type_match_size(typeclass, size, datatype, ierror)
    INTEGER, INTENT(IN) ::  typeclass, size
    TYPE(MPI_Datatype), INTENT(OUT) ::  datatype
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

### A.3.13 Tools / Profiling Interface Fortran 2008 Bindings

```
MPI_Pcontrol(level)
    INTEGER, INTENT(IN) ::  level
```

## A.4 Fortran Bindings with mpif.h or the mpi Module

### A.4.1 Point-to-Point Communication Fortran Bindings

**F binding**

```
MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

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

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

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
    INTEGER(KIND=MPI_ADDRESS_KIND) BUFFER_ADDR
    INTEGER SIZE, IERROR

MPI_CANCEL(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

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

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

MPI_IMPROBE(SOURCE, TAG, COMM, FLAG, MESSAGE, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

MPI_IMRECV(BUF, COUNT, DATATYPE, MESSAGE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, MESSAGE, REQUEST, IERROR

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
    INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

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

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

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

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

MPI_MRECV(BUF, COUNT, DATATYPE, MESSAGE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, MESSAGE, STATUS(MPI_STATUS_SIZE), IERROR

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

MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR

MPI_RECV_INIT(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI_REQUEST_FREE(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

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

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

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

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

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
    SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG,
             COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM,
    STATUS(MPI_STATUS_SIZE), IERROR
```

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

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

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

MPI_START(REQUEST, IERROR)
    INTEGER REQUEST, IERROR

MPI_STARTALL(COUNT, ARRAY_OF_REQUESTS, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), IERROR

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG, ARRAY_OF_STATUSES, IERROR)
    INTEGER COUNT, ARRAY_OF_REQUESTS(*),
    ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
    LOGICAL FLAG

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

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_TEST_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

MPI_WAIT(REQUEST, STATUS, IERROR)
    INTEGER REQUEST, STATUS(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_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.4.2 Datatypes Fortran Bindings

```
INTEGER(KIND=MPI_ADDRESS_KIND) MPI_AINT_ADD(BASE, DISP)
    INTEGER(KIND=MPI_ADDRESS_KIND) BASE, DISP

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

MPI_GET_ADDRESS(LOCATION, ADDRESS, IERROR)
    <type> LOCATION(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) ADDRESS
    INTEGER IERROR

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR

MPI_PACK_EXTERNAL(DATAREP, INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE,
              POSITION, IERROR)
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) OUTSIZE, POSITION

MPI_PACK_EXTERNAL_SIZE(DATAREP, INCOUNT, DATATYPE, SIZE, IERROR)
    CHARACTER*(*) DATAREP
    INTEGER INCOUNT, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE

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_DARGS, ARRAY_OF_PSIZES, ORDER,
              OLDTYPE, NEWTYPE, IERROR)
    INTEGER SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
    ARRAY_OF_DARGS(*), 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_HINDEXED_BLOCK(COUNT, BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
             OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, OLDTYPE, NEWTYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_DISPLACEMENTS(*)

MPI_TYPE_CREATE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE,
             IERROR)
    INTEGER COUNT, BLOCKLENGTH, 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,
    IERROR
    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(OLDTYPE, NEWTYPE, IERROR)
    INTEGER OLDTYPE, 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_DATATYPES(*), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ARRAY_OF_ADDRESSES(*)

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

MPI_TYPE_GET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) COUNT

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_EXTENT_X(DATATYPE, LB, EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_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_GET_TRUE_EXTENT_X(DATATYPE, TRUE_LB, TRUE_EXTENT, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_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_SIZE_X(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) SIZE

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCOUNT, DATATYPE, COMM,
             IERROR)
    <type> INBUF(*), OUTBUF(*)
    INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

MPI_UNPACK_EXTERNAL(DATAREP, INBUF, INSIZE, POSITION, OUTBUF, OUTSIZE,
             DATATYPE, IERROR)
    CHARACTER*(*) DATAREP
    <type> INBUF(*), OUTBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) INSIZE, POSITION
    INTEGER OUTSIZE, DATATYPE, IERROR
```

### A.4.3 Collective Communication Fortran Bindings

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

```
MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              IERROR

MPI_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              INFO, REQUEST, IERROR

MPI_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR

MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR

MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, IERROR

MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, INFO, REQUEST, IERROR

MPI_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
              RDISPLS(*), RECVTYPES(*), COMM, IERROR

MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
```

```
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
              RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR

MPI_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR

MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR

MPI_BARRIER_INIT(COMM, INFO, REQUEST, IERROR)
    INTEGER COMM, INFO, REQUEST, IERROR

MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

MPI_BCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR

MPI_EXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR

MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
              COMM, IERROR

MPI_GATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
              COMM, INFO, REQUEST, IERROR

MPI_GATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
              REQUEST, IERROR

MPI_IALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

MPI_IALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              REQUEST, IERROR

MPI_IALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_IALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

MPI_IALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF, RECVCOUNTS,
             RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, REQUEST, IERROR

MPI_IALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
              RDISPLS(*), RECVTYPES(*), COMM, REQUEST, IERROR

MPI_IBARRIER(COMM, REQUEST, IERROR)
    INTEGER COMM, REQUEST, IERROR

MPI_IBCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, REQUEST, IERROR

MPI_IEXSCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_IGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
              IERROR
```

```
MPI_IGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
              COMM, REQUEST, IERROR

MPI_IREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, REQUEST, IERROR

MPI_IREDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, REQUEST, IERROR

MPI_IREDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_ISCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, REQUEST, IERROR

MPI_ISCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, REQUEST,
              IERROR

MPI_ISCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
              COMM, REQUEST, IERROR

MPI_OP_COMMUTATIVE(OP, COMMUTE, IERROR)
    LOGICAL COMMUTE
    INTEGER OP, IERROR

MPI_OP_CREATE( USER_FN, COMMUTE, OP, IERROR)
    EXTERNAL USER_FN
    LOGICAL COMMUTE
    INTEGER OP, IERROR

MPI_OP_FREE(OP, IERROR)
    INTEGER OP, IERROR

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

```
MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,
             REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
```

```
MPI_REDUCE_LOCAL(INBUF, INOUTBUF, COUNT, DATATYPE, OP, IERROR)
    <type> INBUF(*), INOUTBUF(*)
    INTEGER COUNT, DATATYPE, OP, IERROR
```

```
MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR
```

```
MPI_REDUCE_SCATTER_BLOCK(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP, COMM,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, IERROR
```

```
MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
             COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

```
MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
             INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

```
MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, IERROR
```

```
MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

```
MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

```
MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
              COMM, IERROR
```

```
MPI_SCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
```

```
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
              COMM, INFO, REQUEST, IERROR

MPI_SCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
              REQUEST, IERROR
```

### A.4.4 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_GROUP(COMM, GROUP, TAG, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, TAG, NEWCOMM, IERROR

MPI_COMM_CREATE_KEYVAL(COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN, COMM_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL COMM_COPY_ATTR_FN, COMM_DELETE_ATTR_FN
    INTEGER COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_COMM_DELETE_ATTR(COMM, COMM_KEYVAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR

MPI_COMM_DUP_FN(OLDCOMM, COMM_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDCOMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
              ATTRIBUTE_VAL_OUT
    LOGICAL FLAG

MPI_COMM_DUP_WITH_INFO(COMM, INFO, NEWCOMM, IERROR)
    INTEGER COMM, INFO, NEWCOMM, IERROR

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_INFO(COMM, INFO_USED, IERROR)
    INTEGER COMM, INFO_USED, IERROR

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_IDUP(COMM, NEWCOMM, REQUEST, IERROR)
    INTEGER COMM, NEWCOMM, REQUEST, IERROR

MPI_COMM_IDUP_WITH_INFO(COMM, INFO, NEWCOMM, REQUEST, IERROR)
    INTEGER COMM, INFO, NEWCOMM, REQUEST, IERROR

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

MPI_COMM_SET_ATTR(COMM, COMM_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, COMM_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_COMM_SET_INFO(COMM, INFO, IERROR)
    INTEGER COMM, INFO, IERROR

MPI_COMM_SET_NAME(COMM, COMM_NAME, IERROR)
    INTEGER COMM, IERROR
    CHARACTER*(*) COMM_NAME

MPI_COMM_SIZE(COMM, SIZE, IERROR)
    INTEGER COMM, SIZE, IERROR

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
    INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR)
```

```
    INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR

MPI_COMM_TEST_INTER(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG

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,
              TAG, NEWINTERCOMM, IERROR)
    INTEGER LOCAL_COMM, LOCAL_LEADER, PEER_COMM, REMOTE_LEADER, TAG,
    NEWINTERCOMM, IERROR

MPI_INTERCOMM_MERGE(INTERCOMM, HIGH, NEWINTRACOMM, IERROR)
    INTEGER INTERCOMM, NEWINTRACOMM, 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(DATATYPE, TYPE_KEYVAL, IERROR)
    INTEGER DATATYPE, 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(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG

MPI_TYPE_GET_NAME(DATATYPE, TYPE_NAME, RESULTLEN, IERROR)
    INTEGER DATATYPE, 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(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE,
             IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE

MPI_TYPE_SET_ATTR(DATATYPE, TYPE_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER DATATYPE, TYPE_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL

MPI_TYPE_SET_NAME(DATATYPE, TYPE_NAME, IERROR)
    INTEGER DATATYPE, IERROR
    CHARACTER*(*) TYPE_NAME

MPI_WIN_CREATE_KEYVAL(WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN, WIN_KEYVAL,
             EXTRA_STATE, IERROR)
    EXTERNAL WIN_COPY_ATTR_FN, WIN_DELETE_ATTR_FN
    INTEGER WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

MPI_WIN_DELETE_ATTR(WIN, WIN_KEYVAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
```

```
MPI_WIN_DUP_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
              ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

```
MPI_WIN_FREE_KEYVAL(WIN_KEYVAL, IERROR)
    INTEGER WIN_KEYVAL, IERROR
```

```
MPI_WIN_GET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
    LOGICAL FLAG
```

```
MPI_WIN_GET_NAME(WIN, WIN_NAME, RESULTLEN, IERROR)
    INTEGER WIN, RESULTLEN, IERROR
    CHARACTER*(*) WIN_NAME
```

```
MPI_WIN_NULL_COPY_FN(OLDWIN, WIN_KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT, FLAG, IERROR)
    INTEGER OLDWIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE, ATTRIBUTE_VAL_IN,
              ATTRIBUTE_VAL_OUT
    LOGICAL FLAG
```

```
MPI_WIN_NULL_DELETE_FN(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, EXTRA_STATE, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL, EXTRA_STATE
```

```
MPI_WIN_SET_ATTR(WIN, WIN_KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER WIN, WIN_KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) ATTRIBUTE_VAL
```

```
MPI_WIN_SET_NAME(WIN, WIN_NAME, IERROR)
    INTEGER WIN, IERROR
    CHARACTER*(*) WIN_NAME
```

### A.4.5 Process Topologies Fortran Bindings

```
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

```
MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR
```

```
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```

```
MPI_CART_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_DIST_GRAPH_CREATE(COMM_OLD, N, SOURCES, DEGREES, DESTINATIONS, WEIGHTS,
             INFO, REORDER, COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, N, SOURCES(*), DEGREES(*), DESTINATIONS(*),
    WEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER

MPI_DIST_GRAPH_CREATE_ADJACENT(COMM_OLD, INDEGREE, SOURCES, SOURCEWEIGHTS,
             OUTDEGREE, DESTINATIONS, DESTWEIGHTS, INFO, REORDER,
             COMM_DIST_GRAPH, IERROR)
    INTEGER COMM_OLD, INDEGREE, SOURCES(*), SOURCEWEIGHTS(*), OUTDEGREE,
    DESTINATIONS(*), DESTWEIGHTS(*), INFO, COMM_DIST_GRAPH, IERROR
    LOGICAL REORDER

MPI_DIST_GRAPH_NEIGHBORS(COMM, MAXINDEGREE, SOURCES, SOURCEWEIGHTS,
             MAXOUTDEGREE, DESTINATIONS, DESTWEIGHTS, IERROR)
    INTEGER COMM, MAXINDEGREE, SOURCES(*), SOURCEWEIGHTS(*), MAXOUTDEGREE,
    DESTINATIONS(*), DESTWEIGHTS(*), IERROR

MPI_DIST_GRAPH_NEIGHBORS_COUNT(COMM, INDEGREE, OUTDEGREE, WEIGHTED, IERROR)
    INTEGER COMM, INDEGREE, OUTDEGREE, IERROR
    LOGICAL WEIGHTED

MPI_GRAPHDIMS_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_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR

MPI_GRAPH_MAP(COMM, NNODES, INDEX, EDGES, NEWRANK, IERROR)
    INTEGER COMM, NNODES, INDEX(*), EDGES(*), NEWRANK, IERROR
```

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR

MPI_INEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

MPI_INEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              REQUEST, IERROR

MPI_INEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, REQUEST, IERROR

MPI_INEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, REQUEST, IERROR

MPI_INEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
              REQUEST, IERROR

MPI_NEIGHBOR_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

MPI_NEIGHBOR_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
             DISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    IERROR

MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
              INFO, REQUEST, IERROR
```

```
MPI_NEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR

MPI_NEIGHBOR_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR

MPI_NEIGHBOR_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, IERROR

MPI_NEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
             RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
              RECVTYPE, COMM, INFO, REQUEST, IERROR

MPI_NEIGHBOR_ALLTOALLW(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
             RECVCOUNTS, RDISPLS, RECVTYPES, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
              IERROR

MPI_NEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
             RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
             IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
              INFO, REQUEST, IERROR

MPI_NEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
              IERROR

MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

### A.4.6 MPI Environmental 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(KIND=MPI_ADDRESS_KIND SIZE
    INTEGER INFO, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) BASEPTR

MPI_COMM_CALL_ERRHANDLER(COMM, ERRORCODE, IERROR)
    INTEGER COMM, ERRORCODE, IERROR

MPI_COMM_CREATE_ERRHANDLER(COMM_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL COMM_ERRHANDLER_FN
    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(FILE_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL FILE_ERRHANDLER_FN
    INTEGER ERRHANDLER, IERROR

MPI_FILE_GET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR

MPI_FILE_SET_ERRHANDLER(FILE, ERRHANDLER, IERROR)
    INTEGER FILE, ERRHANDLER, IERROR
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR

MPI_FINALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_FREE_MEM(BASE, IERROR)
    <type> BASE(*)
    INTEGER IERROR

MPI_GET_LIBRARY_VERSION(VERSION, RESULTLEN, IERROR)
    CHARACTER*(*) VERSION
    INTEGER RESULTLEN, IERROR

MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
    CHARACTER*(*) NAME
    INTEGER RESULTLEN, IERROR

MPI_GET_VERSION(VERSION, SUBVERSION, IERROR)
    INTEGER VERSION, SUBVERSION, IERROR

MPI_INIT(IERROR)
    INTEGER IERROR

MPI_INITIALIZED(FLAG, IERROR)
    LOGICAL FLAG
    INTEGER IERROR

MPI_WIN_CALL_ERRHANDLER(WIN, ERRORCODE, IERROR)
    INTEGER WIN, ERRORCODE, IERROR

MPI_WIN_CREATE_ERRHANDLER(WIN_ERRHANDLER_FN, ERRHANDLER, IERROR)
    EXTERNAL WIN_ERRHANDLER_FN
    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.4.7 The Info Object Fortran Bindings

```
MPI_INFO_CREATE(INFO, IERROR)
    INTEGER INFO, IERROR

MPI_INFO_DELETE(INFO, KEY, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY

MPI_INFO_DUP(INFO, NEWINFO, IERROR)
    INTEGER INFO, NEWINFO, IERROR
```

```
MPI_INFO_FREE(INFO, IERROR)
    INTEGER INFO, IERROR

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
    CHARACTER*(*) KEY
    LOGICAL FLAG

MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) KEY, VALUE
```

### A.4.8 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_JOIN(FD, INTERCOMM, IERROR)
    INTEGER FD, INTERCOMM, IERROR

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

```
    INTEGER MAXPROCS, INFO, ROOT, COMM, INTERCOMM, ARRAY_OF_ERRCODES(*),
    IERROR

MPI_COMM_SPAWN_MULTIPLE(COUNT, ARRAY_OF_COMMANDS, ARRAY_OF_ARGV,
             ARRAY_OF_MAXPROCS, ARRAY_OF_INFO, ROOT, COMM, INTERCOMM,
             ARRAY_OF_ERRCODES, IERROR)
    INTEGER COUNT, ARRAY_OF_MAXPROCS(*), ARRAY_OF_INFO(*), ROOT, COMM,
    INTERCOMM, ARRAY_OF_ERRCODES(*), IERROR
    CHARACTER*(*) ARRAY_OF_COMMANDS(*), ARRAY_OF_ARGV(*)

MPI_LOOKUP_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR

MPI_OPEN_PORT(INFO, PORT_NAME, IERROR)
    INTEGER INFO, IERROR
    CHARACTER*(*) PORT_NAME

MPI_PUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR

MPI_UNPUBLISH_NAME(SERVICE_NAME, INFO, PORT_NAME, IERROR)
    CHARACTER*(*) SERVICE_NAME, PORT_NAME
    INTEGER INFO, IERROR
```

### A.4.9 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_COMPARE_AND_SWAP(ORIGIN_ADDR, COMPARE_ADDR, RESULT_ADDR, DATATYPE,
             TARGET_RANK, TARGET_DISP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), COMPARE_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER DATATYPE, TARGET_RANK, WIN, IERROR

MPI_FETCH_AND_OP(ORIGIN_ADDR, RESULT_ADDR, DATATYPE, TARGET_RANK,
             TARGET_DISP, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER DATATYPE, TARGET_RANK, 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_GET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_ADDR,
             RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK, TARGET_DISP,
             TARGET_COUNT, TARGET_DATATYPE, OP, WIN, IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, 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_RACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
              TARGET_DATATYPE, OP, WIN, REQUEST, IERROR

MPI_RGET(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
              TARGET_DATATYPE, WIN, REQUEST, IERROR

MPI_RGET_ACCUMULATE(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE,
             RESULT_ADDR, RESULT_COUNT, RESULT_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*), RESULT_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, RESULT_COUNT, RESULT_DATATYPE,
              TARGET_RANK, TARGET_COUNT, TARGET_DATATYPE, OP, WIN, REQUEST,
    IERROR

MPI_RPUT(ORIGIN_ADDR, ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK,
             TARGET_DISP, TARGET_COUNT, TARGET_DATATYPE, WIN, REQUEST,
             IERROR)
    <type> ORIGIN_ADDR(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) TARGET_DISP
    INTEGER ORIGIN_COUNT, ORIGIN_DATATYPE, TARGET_RANK, TARGET_COUNT,
```

```
            TARGET_DATATYPE, WIN, REQUEST, IERROR
```

```
MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

If the Fortran compiler provides `TYPE(C_PTR)`, then overloaded by:

```
  INTERFACE MPI_WIN_ALLOCATE
    SUBROUTINE MPI_WIN_ALLOCATE(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
          WIN, IERROR)
      IMPORT :: MPI_ADDRESS_KIND
      INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_CPTR(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, &
          WIN, IERROR)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
      IMPORT :: MPI_ADDRESS_KIND
      INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
      TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
  END INTERFACE
```

```
MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR, WIN, IERROR)
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

If the Fortran compiler provides `TYPE(C_PTR)`, then overloaded by:

```
  INTERFACE MPI_WIN_ALLOCATE_SHARED
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, &
          BASEPTR, WIN, IERROR)
      IMPORT :: MPI_ADDRESS_KIND
      INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_ALLOCATE_SHARED_CPTR(SIZE, DISP_UNIT, INFO, COMM, &
          BASEPTR, WIN, IERROR)
      USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR
      IMPORT :: MPI_ADDRESS_KIND
      INTEGER :: DISP_UNIT, INFO, COMM, WIN, IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) :: SIZE
      TYPE(C_PTR) :: BASEPTR
    END SUBROUTINE
  END INTERFACE
```

```
MPI_WIN_ATTACH(WIN, BASE, SIZE, IERROR)
    INTEGER WIN, IERROR
    <type> BASE(*)
    INTEGER (KIND=MPI_ADDRESS_KIND) SIZE
```

```
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
    INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR

MPI_WIN_CREATE_DYNAMIC(INFO, COMM, WIN, IERROR)
    INTEGER INFO, COMM, WIN, IERROR

MPI_WIN_DETACH(WIN, BASE, IERROR)
    INTEGER WIN, IERROR
    <type> BASE(*)

MPI_WIN_FENCE(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR

MPI_WIN_FLUSH(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR

MPI_WIN_FLUSH_ALL(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_FLUSH_LOCAL(RANK, WIN, IERROR)
    INTEGER RANK, WIN, IERROR

MPI_WIN_FLUSH_LOCAL_ALL(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_FREE(WIN, IERROR)
    INTEGER WIN, IERROR

MPI_WIN_GET_GROUP(WIN, GROUP, IERROR)
    INTEGER WIN, GROUP, IERROR

MPI_WIN_GET_INFO(WIN, INFO_USED, IERROR)
    INTEGER WIN, INFO_USED, IERROR

MPI_WIN_LOCK(LOCK_TYPE, RANK, ASSERT, WIN, IERROR)
    INTEGER LOCK_TYPE, RANK, ASSERT, WIN, IERROR

MPI_WIN_LOCK_ALL(ASSERT, WIN, IERROR)
    INTEGER ASSERT, WIN, IERROR

MPI_WIN_POST(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR

MPI_WIN_SET_INFO(WIN, INFO, IERROR)
    INTEGER WIN, INFO, IERROR

MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, BASEPTR, IERROR)
    INTEGER WIN, RANK, DISP_UNIT, IERROR
    INTEGER (KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

If the Fortran compiler provides `TYPE(C_PTR)`, then overloaded by:

```
  INTERFACE MPI_WIN_SHARED_QUERY
    SUBROUTINE MPI_WIN_SHARED_QUERY(WIN, RANK, SIZE, DISP_UNIT, &
            BASEPTR, IERROR)
      IMPORT ::  MPI_ADDRESS_KIND
      INTEGER ::  WIN, RANK, DISP_UNIT, IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE, BASEPTR
    END SUBROUTINE
    SUBROUTINE MPI_WIN_SHARED_QUERY_CPTR(WIN, RANK, SIZE, DISP_UNIT, &
            BASEPTR, IERROR)
      USE, INTRINSIC ::  ISO_C_BINDING, ONLY : C_PTR
      IMPORT ::  MPI_ADDRESS_KIND
      INTEGER ::  WIN, RANK, DISP_UNIT, IERROR
      INTEGER(KIND=MPI_ADDRESS_KIND) ::  SIZE
      TYPE(C_PTR) ::  BASEPTR
    END SUBROUTINE
  END INTERFACE
```

```
MPI_WIN_START(GROUP, ASSERT, WIN, IERROR)
    INTEGER GROUP, ASSERT, WIN, IERROR
```

```
MPI_WIN_SYNC(WIN, IERROR)
    INTEGER 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_UNLOCK_ALL(WIN, IERROR)
    INTEGER WIN, IERROR
```

```
MPI_WIN_WAIT(WIN, IERROR)
    INTEGER WIN, IERROR
```

### A.4.10 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)
    EXTERNAL QUERY_FN, FREE_FN, CANCEL_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER REQUEST, IERROR
```

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
    INTEGER REQUIRED, PROVIDED, IERROR
```

```
MPI_IS_THREAD_MAIN(FLAG, IERROR)
    LOGICAL FLAG
```

```
    INTEGER IERROR

MPI_QUERY_THREAD(PROVIDED, IERROR)
    INTEGER PROVIDED, IERROR

MPI_STATUS_SET_CANCELLED(STATUS, FLAG, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG

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

MPI_STATUS_SET_ELEMENTS_X(STATUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, IERROR
    INTEGER(KIND=MPI_COUNT_KIND) COUNT
```

### A.4.11 I/O Fortran Bindings

```
MPI_CONVERSION_FN_NULL(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

MPI_FILE_CLOSE(FH, IERROR)
    INTEGER FH, IERROR

MPI_FILE_DELETE(FILENAME, INFO, IERROR)
    CHARACTER*(*) FILENAME
    INTEGER INFO, IERROR

MPI_FILE_GET_AMODE(FH, AMODE, IERROR)
    INTEGER FH, AMODE, IERROR

MPI_FILE_GET_ATOMICITY(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG

MPI_FILE_GET_BYTE_OFFSET(FH, OFFSET, DISP, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET, DISP

MPI_FILE_GET_GROUP(FH, GROUP, IERROR)
    INTEGER FH, GROUP, IERROR

MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
    INTEGER FH, INFO_USED, IERROR

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

```
MPI_FILE_GET_POSITION_SHARED(FH, OFFSET, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_GET_SIZE(FH, SIZE, IERROR)
    INTEGER FH, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) SIZE

MPI_FILE_GET_TYPE_EXTENT(FH, DATATYPE, EXTENT, IERROR)
    INTEGER FH, DATATYPE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTENT

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

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

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

MPI_FILE_IREAD_AT(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

MPI_FILE_IREAD_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

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

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

MPI_FILE_IWRITE_ALL(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_AT_ALL(FH, OFFSET, BUF, COUNT, DATATYPE, REQUEST, IERROR)
```

```
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, REQUEST, IERROR
    INTEGER(KIND=MPI_OFFSET_KIND) OFFSET

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(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, 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_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_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_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_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_ORDERED(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_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(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
    <type> BUF(*)
    INTEGER FH, COUNT, DATATYPE, 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_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_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_WRITE_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_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, 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_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, 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)
    CHARACTER*(*) DATAREP
    EXTERNAL READ_CONVERSION_FN, WRITE_CONVERSION_FN, DTYPE_FILE_EXTENT_FN
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE
    INTEGER IERROR
```

### A.4.12 Language Bindings Fortran Bindings

```
MPI_F_SYNC_REG(BUF)
    <type> BUF(*)

MPI_SIZEOF(X, SIZE, IERROR)
    <type> X(*)
```

```
    INTEGER SIZE, IERROR

MPI_STATUS_F082F(F08_STATUS, F_STATUS, IERROR)
    TYPE(MPI_Status) ::  F08_STATUS
    INTEGER ::  F_STATUS(MPI_STATUS_SIZE), IERROR

MPI_STATUS_F2F08(F_STATUS, F08_STATUS, IERROR)
    INTEGER ::  F_STATUS(MPI_STATUS_SIZE), IERROR
    TYPE(MPI_Status) ::  F08_STATUS

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, DATATYPE, IERROR)
    INTEGER TYPECLASS, SIZE, DATATYPE, IERROR
```

### A.4.13 Tools / Profiling Interface Fortran Bindings

```
MPI_PCONTROL(LEVEL)
    INTEGER LEVEL
```

### A.4.14 Deprecated Fortran Bindings

```
MPI_ATTR_DELETE(COMM, KEYVAL, IERROR)
    INTEGER COMM, KEYVAL, IERROR

MPI_ATTR_GET(COMM, KEYVAL, ATTRIBUTE_VAL, FLAG, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR
    LOGICAL FLAG

MPI_ATTR_PUT(COMM, KEYVAL, ATTRIBUTE_VAL, IERROR)
    INTEGER COMM, KEYVAL, ATTRIBUTE_VAL, IERROR

MPI_DUP_FN(OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
             ATTRIBUTE_VAL_OUT, FLAG, IERR)
    INTEGER OLDCOMM, KEYVAL, EXTRA_STATE, ATTRIBUTE_VAL_IN,
    ATTRIBUTE_VAL_OUT, IERR
    LOGICAL FLAG

MPI_KEYVAL_CREATE(COPY_FN, DELETE_FN, KEYVAL, EXTRA_STATE, IERROR)
    EXTERNAL COPY_FN, DELETE_FN
    INTEGER KEYVAL, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) EXTRA_STATE

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

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
```

# Annex B

# Change-Log

Annex B.1 summarizes changes from the previous version of the MPI standard to the version presented by this document. Only significant changes (i.e., clarifications and new features) that might either require implementation effort in the MPI libraries or change the understanding of MPI from a user's perspective are presented. Editorial modifications, formatting, typo corrections and minor clarifications are not shown. If not otherwise noted, the section and page references refer to the locations of the change or new functionality in this version of the standard. Changes in Annexes B.2–B.5 were already introduced in the corresponding sections in previous versions of this standard.

## B.1 Changes from Version 3.1 to Version 3.2

### B.1.1 Changes in MPI-3.2

1. Section 3.8.4 on page 76.
   Cancelling a send request by calling MPI_CANCEL has been deprecated and may be removed in a future version of the MPI specification.

2. Sections 3.7.3, 3.9, 5.13, 7.8, and 7.9 on pages 55, 77, 223, 363, and 369.
   Persistent collective communication and persistent neighborhood communication are added to the standard.

3. Section 6.4.2 on page 266, and MPI-3.1 Section 6.4.2 on page 237.
   The functions MPI_COMM_DUP and MPI_COMM_IDUP were updated to no longer propagate info hints.
   This change may affect backward compatibility.

4. Sections 6.4.4, 11.2.7, and 13.2.8 on pages 279, 462, and 551, and
   MPI-3.1 Sections 6.4.4, 11.2.7, and 13.2.8 on pages 248, 415, and 500.
   The definition of info hints was updated to allow applications to provide assertions regarding their usage of MPI objects and operations.

5. Section 6.4.4 on page 279.
   The new info hints mpi_assert_no_any_tag, mpi_assert_no_any_source, mpi_assert_exact_length, and mpi_assert_allow_overtaking were added for use with communicators.

6. Section 6.4.2 on page 266.
The MPI_COMM_IDUP_WITH_INFO function was added.

7. Sections 6.4.4, 11.2.7, and 13.2.8 on pages 279, 462, and 551.
The semantics of the MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, MPI_WIN_GET_INFO, MPI_FILE_SET_INFO, and MPI_FILE_GET_INFO were clarified.

8. Section 7.5.
MPI_DIMS_CREATE is now guaranteed to return MPI_SUCCESS if the number of dimensions passed to the routine is set to 0 and the number of nodes is set to 1.

9. Sections 2.8, 8.3, 8.5, and 8.7 on pages 20, 382, 394, and 399.
MPI calls that are not related to any objects are considered to be attached to the communicator MPI_COMM_SELF instead of MPI_COMM_WORLD. The definition of MPI_ERRORS_ARE_FATAL was clarified to cover all connected processes, and a new error handler, MPI_ERRORS_ABORT, was created to limit the scope of aborting.

10. Section 12.3 on page **??**
The `mpi_f08` binding incorrectly had the dummy parameter `flag` in the MPI F08 binding for MPI_SET_STATUS_CANCELLED marked as `INTENT(OUT)`. It has been fixed to be `INTENT(IN)`.

11. Sections 8.3 and 8.4 on pages 382 and 391.
Clarified definition of errors to say that MPI should continue whenever possible and allow the user to recover from errors.

## B.2 Changes from Version 3.0 to Version 3.1

### B.2.1 Fixes to Errata in Previous Versions of MPI

1. Chapters 3–17, Annex A.3 on page 732, and Example 5.21 on page 195, and MPI-3.0 Chapters 3-17, Annex A.3 on page 707, and Example 5.21 on page 187.
Within the `mpi_f08` Fortran support method, `BIND(C)` was removed from all `SUBROUTINE`, `FUNCTION`, and `ABSTRACT INTERFACE` definitions.

2. Section 3.2.5 on page 32, and MPI-3.0 Section 3.2.5 on page 30.
The three public fields MPI_SOURCE, MPI_TAG, and MPI_ERROR of the Fortran derived type `TYPE(MPI_Status)` must be of type `INTEGER`.

3. Section 3.8.2 on page 71, and MPI-3.0 Section 3.8.2 on page 67.
The flag arguments of the Fortran interfaces of MPI_IMPROBE were originally incorrectly defined as `INTEGER` (instead as `LOGICAL`).

4. Section 6.4.2 on page 266, and MPI-3.0 Section 6.4.2 on page 237.
In the `mpi_f08` binding of MPI_COMM_IDUP, the output argument newcomm is declared as `ASYNCHRONOUS`.

5. Section 6.4.4 on page 279, and MPI-3.0 Section 6.4.4 on page 248.
In the `mpi_f08` binding of MPI_COMM_SET_INFO, the intent of comm is `IN`, and the optional output argument ierror was missing.

6. Section 7.6 on page 348, and MPI-3.0 Sections 7.6, on pages 314.
   In the case of virtual general graph topolgies (created with MPI_CART_CREATE), the use of neighborhood collective communication is restricted to adjacency matrices with the number of edges between any two processes is defined to be the same for both processes (i.e., with a symmetric adjacency matrix).

7. Section 8.1.1 on page 375, and MPI-3.0 Section 8.1.1 on page 335.
   In the mpi_f08 binding of MPI_GET_LIBRARY_VERSION, a typo in the resultlen argument was corrected.

8. Sections 8.2 (MPI_ALLOC_MEM and MPI_ALLOC_MEM_CPTR),
   11.2.2 (MPI_WIN_ALLOCATE and MPI_WIN_ALLOCATE_CPTR),
   11.2.3 (MPI_WIN_ALLOCATE_SHARED and MPI_WIN_ALLOCATE_SHARED_CPTR),
   11.2.3 (MPI_WIN_SHARED_QUERY and MPI_WIN_SHARED_QUERY_CPTR),
   **??** and **??** (Profiling interface), and corresponding sections in MPI-3.0.
   The linker name concept was substituted by defining specific procedure names.

9. Section 11.2.1 on page 449, and MPI-3.0 Section 11.2.2 on page 407.
   The same_size info key can be used with all window flavors, and requires that all processes in the process group of the communicator have provided this info key with the same value.

10. Section 11.3.4 on page 470, and MPI-3.0 Section 11.3.4 on page 424.
    Origin buffer arguments to MPI_GET_ACCUMULATE are ignored when the MPI_NO_OP operation is used.

11. Section 11.3.4 on page 470, and MPI-3.0 Section 11.3.4 on page 424.
    Clarify the roles of origin, result, and target communication parameters in MPI_GET_ACCUMULATE.

12. Section **??** on page **??**, and MPI-3.0 Section 14.3 on page 561
    New paragraph and advice to users clarifying intent of variable names in the tools information interface.

13. Section **??** on page **??**, and MPI-3.0 Section 14.3.3 on page 563.
    New paragraph clarifying variable name equivalence in the tools information interface.

14. Sections **??**, **??**, and **??** on pages **??**, **??**, and **??**, and
    MPI-3.0 Sections 14.3.6, 14.3.7, and 14.3.8 on pages 567, 573, and 584.
    In functions MPI_T_CVAR_GET_INFO, MPI_T_PVAR_GET_INFO, and MPI_T_CATEGORY_GET_INFO, clarification of parameters that must be identical for equivalent control variable / performance variable / category names across connected processes.

15. Section **??** on page **??**, and MPI-3.0 Section 14.3.7 on page 573.
    Clarify return code of MPI_T_PVAR_{START,STOP,RESET} routines.

16. Section **??** on page **??**, and MPI-3.0 Section 14.3.7 on page 579, line 7.
    Clarify the return code when bad handle is passed to an MPI_T_PVAR_* routine.

17. Section 17.1.4 on page 627, and MPI-3.0 Section 17.1.4 on page 603.
    The advice to implementors at the end of the section was rewritten and moved into the following section.

18. Section 17.1.5 on page 628, and MPI-3.0 Section 17.1.5 on page 605.
The section was fully rewritten. The linker name concept was substituted by defining specific procedure names.

19. Section 17.1.6 on page 633, and MPI-3.0 Section 17.1.6 on page 611.
The requirements on `BIND(C)` procedure interfaces were removed.

20. Annexes A.2, A.3, and A.4 on pages 708, 732, and 787, and
MPI-3.0 Annexes A.2, A.3, and A.4 on pages 685, 707, and 756.
The predefined callback MPI_CONVERSION_FN_NULL was added to all three annexes.

21. Annex A.3.4 on page 753, and MPI-3.0 Annex A.3.4 on page 724.
In the `mpi_f08` binding of
MPI_{COMM|TYPE|WIN}_{DUP|NULL_COPY|NULL_DELETE}_FN, all `INTENT(...)` information was removed.

### B.2.2 Changes in MPI-3.1

1. Sections 2.6.4 and 4.1.5 on pages 20 and 106.
The use of the intrinsic operators "+" and "-" for absolute addresses is substituted by MPI_AINT_ADD and MPI_AINT_DIFF. In C, they can be implemented as macros.

2. Sections 8.1.1, 8.7, and 12.4 on pages 375, 399, and 532.
The routines MPI_INITIALIZED, MPI_FINALIZED, MPI_QUERY_THREAD, MPI_IS_THREAD_MAIN, MPI_GET_VERSION, and MPI_GET_LIBRARY_VERSION are callable from threads without restriction (in the sense of MPI_THREAD_MULTIPLE), irrespective of the actual level of thread support provided, in the case where the implementation supports threads.

3. Section 11.2.1 on page 449.
The same_disp_unit info key was added for use in RMA window creation routines.

4. Sections 13.4.2 and 13.4.3 on pages 560 and 565.
Added MPI_FILE_IREAD_AT_ALL, MPI_FILE_IWRITE_AT_ALL, MPI_FILE_IREAD_ALL, and MPI_FILE_IWRITE_ALL

5. Sections **??**, **??**, and **??** on pages **??**, **??**, and **??**.
Clarified that `NULL` parameters can be provided in MPI_T_{CVAR|PVAR|CATEGORY}_GET_INFO routines.

6. Sections **??**, **??**, **??**, and **??** on pages **??**, **??**, **??**, and **??**.
New routines MPI_T_CVAR_GET_INDEX, MPI_T_PVAR_GET_INDEX, MPI_T_CATEGORY_GET_INDEX, were added to support retrieving indices of variables and categories. The error codes MPI_T_ERR_INVALID and MPI_T_ERR_INVALID_NAME were added to indicate invalid uses of the interface.

## B.3 Changes from Version 2.2 to Version 3.0

### B.3.1 Fixes to Errata in Previous Versions of MPI

1. Sections 2.6.2 and 2.6.3 on pages 19 and 19, and MPI-2.2 Section 2.6.2 on page 17, lines 41-42, Section 2.6.3 on page 18, lines 15-16, and Section 2.6.4 on page 18, lines 40-41.
   This is an MPI-2 erratum: The scope for the reserved prefix `MPI_` and the C++ namespace `MPI` is now any name as originally intended in MPI-1.

2. Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, and Annex A.1.1 on pages 27, 184, 592, and 685, and MPI-2.2 Sections 3.2.2, 5.9.2, 13.5.2 Table 13.2, 16.1.16 Table 16.1, and Annex A.1.1 on pages 27, 164, 433, 472 and 513
   This is an MPI-2.2 erratum: New named predefined datatypes MPI_CXX_BOOL, MPI_CXX_FLOAT_COMPLEX, MPI_CXX_DOUBLE_COMPLEX, and MPI_CXX_LONG_DOUBLE_COMPLEX were added in C and Fortran corresponding to the C++ types `bool`, `std::complex<float>`, `std::complex<double>`, and `std::complex<long double>`. These datatypes also correspond to the deprecated C++ predefined datatypes MPI::BOOL, MPI::COMPLEX, MPI::DOUBLE_COMPLEX, and MPI::LONG_DOUBLE_COMPLEX, which were removed in MPI-3.0. The non-standard C++ types `Complex<...>` were substituted by the standard types `std::complex<...>`.

3. Sections 5.9.2 on pages 184 and MPI-2.2 Section 5.9.2, page 165, line 47.
   This is an MPI-2.2 erratum: MPI_C_COMPLEX was added to the "Complex" reduction group.

4. Section 7.5.5 on page 335, and MPI-2.2, Section 7.5.5 on page 257, C++ interface on page 264, line 3.
   This is an MPI-2.2 erratum: The argument rank was removed and in/outdegree are now defined as `int& indegree` and `int& outdegree` in the C++ interface of MPI_DIST_GRAPH_NEIGHBORS_COUNT.

5. Section 13.5.2, Table 13.2 on page 592, and MPI-2.2, Section 13.5.3, Table 13.2 on page 433.
   This was an MPI-2.2 erratum: The MPI_C_BOOL "external32" representation is corrected to a 1-byte size.

6. MPI-2.2 Section 16.1.16 on page 471, line 45.
   This is an MPI-2.2 erratum: The constant MPI::_LONG_LONG should be MPI::LONG_LONG.

7. Annex A.1.1 on page 685, Table "Optional datatypes (Fortran)," and MPI-2.2, Annex A.1.1, Table on page 517, lines 34, and 37-41.
   This is an MPI-2.2 erratum: The C++ datatype handles MPI::INTEGER16, MPI::REAL16, MPI::F_COMPLEX4, MPI::F_COMPLEX8, MPI::F_COMPLEX16, MPI::F_COMPLEX32 were added to the table.

### B.3.2 Changes in MPI-3.0

1. Section 2.6.1 on page 17, Section 15.2 on page 618 and all other chapters.
   The C++ bindings were removed from the standard. See errata in Section B.3.1 on page 825 for the latest changes to the MPI C++ binding defined in MPI-2.2.
   This change may affect backward compatibility.

2. Section 2.6.1 on page 17, Section 14.1 on page 613 and Section 15.1 on page 617.
   The deprecated functions MPI_TYPE_HVECTOR, MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT, MPI_ADDRESS, MPI_TYPE_EXTENT, MPI_TYPE_LB, MPI_TYPE_UB, MPI_ERRHANDLER_CREATE (and its callback function prototype MPI_Handler_function), MPI_ERRHANDLER_SET, MPI_ERRHANDLER_GET, the deprecated special datatype handles MPI_LB, MPI_UB, and the constants MPI_COMBINER_HINDEXED_INTEGER, MPI_COMBINER_HVECTOR_INTEGER, MPI_COMBINER_STRUCT_INTEGER were removed from the standard.
   This change may affect backward compatibility.

3. Section 2.3 on page 10.
   Clarified parameter usage for IN parameters. C bindings are now const-correct where backward compatibility is preserved.

4. Section 2.5.4 on page 15 and Section 7.5.4 on page 328.
   The recommended C implementation value for MPI_UNWEIGHTED changed from NULL to non-NULL. An additional weight array constant (MPI_WEIGHTS_EMPTY) was introduced.

5. Section 2.5.4 on page 15 and Section 8.1.1 on page 375.
   Added the new routine MPI_GET_LIBRARY_VERSION to query library specific versions, and the new constant MPI_MAX_LIBRARY_VERSION_STRING.

6. Sections 2.5.8, 3.2.2, 3.3, 5.9.2, on pages 17, 27, 29, 184, Sections 4.1, 4.1.7, 4.1.8, 4.1.11, 12.3 on pages 87, 112, 114, 117, 531, and Annex A.1.1 on page 685.
   New inquiry functions, MPI_TYPE_SIZE_X, MPI_TYPE_GET_EXTENT_X, MPI_TYPE_GET_TRUE_EXTENT_X, and MPI_GET_ELEMENTS_X, return their results as an MPI_Count value, which is a new type large enough to represent element counts in memory, file views, etc. A new function, MPI_STATUS_SET_ELEMENTS_X, modifies the opaque part of an MPI_Status object so that a call to MPI_GET_ELEMENTS_X returns the provided MPI_Count value (in Fortran, `INTEGER (KIND=MPI_COUNT_KIND)`). The corresponding predefined datatype is MPI_COUNT.

7. Chapter 3 on page 25 until Chapter 17 on page 621.
   In the C language bindings, the array-arguments' interfaces were modified to consistently use use `[]` instead of `*`.

   Exceptions are MPI_INIT, which continues to use `char ***argv` (correct because of subtle rules regarding the use of the `&` operator with `char *argv[]`), and MPI_INIT_THREAD, which is changed to be consistent with MPI_INIT.

8. Sections 3.2.5, 4.1.5, 4.1.11, 4.2 on pages 32, 106, 117, 138.
   The functions MPI_GET_COUNT and MPI_GET_ELEMENTS were defined to set the

count argument to MPI_UNDEFINED when that argument would overflow. The functions MPI_PACK_SIZE and MPI_TYPE_SIZE were defined to set the size argument to MPI_UNDEFINED when that argument would overflow. In all other MPI-2.2 routines, the type and semantics of the count arguments remain unchanged, i.e., int or INTEGER.

9. Section 3.2.6 on page 34, and Section 3.8 on page 68.
MPI_STATUS_IGNORE can be also used in MPI_IPROBE, MPI_PROBE, MPI_IMPROBE, and MPI_MPROBE.

10. Section 3.8 on page 68 and Section 3.11 on page 86.
The use of MPI_PROC_NULL in probe operations was clarified. A special predefined message MPI_MESSAGE_NO_PROC was defined for the use of matching probe (i.e., the new MPI_MPROBE and MPI_IMPROBE) with MPI_PROC_NULL.

11. Sections 3.8.2, 3.8.3, 17.2.4, A.1.1 on pages 71, 73, 670, 685.
Like MPI_PROBE and MPI_IPROBE, the new MPI_MPROBE and MPI_IMPROBE operations allow incoming messages to be queried without actually receiving them, except that MPI_MPROBE and MPI_IMPROBE provide a mechanism to receive the specific message with the new routines MPI_MRECV and MPI_IMRECV regardless of other intervening probe or receive operations. The opaque object MPI_Message, the null handle MPI_MESSAGE_NULL, and the conversion functions MPI_Message_c2f and MPI_Message_f2c were defined.

12. Section 4.1.2 on page 89 and Section 4.1.13 on page 122.
The routine MPI_TYPE_CREATE_HINDEXED_BLOCK and constant MPI_COMBINER_HINDEXED_BLOCK were added.

13. Chapter 5 on page 149 and Section 5.12 on page 205.
Added nonblocking interfaces to all collective operations.

14. Sections 6.4.2, 6.4.4, 11.2.7, on pages 266, 279, 462.
The new routines MPI_COMM_DUP_WITH_INFO, MPI_COMM_SET_INFO, MPI_COMM_GET_INFO, MPI_WIN_SET_INFO, and MPI_WIN_GET_INFO were added. The routine MPI_COMM_DUP must also duplicate info hints.

15. Section 6.4.2 on page 266.
Added MPI_COMM_IDUP.

16. Section 6.4.2 on page 266.
Added the new communicator construction routine MPI_COMM_CREATE_GROUP, which is invoked only by the processes in the group of the new communicator being constructed.

17. Section 6.4.2 on page 266.
Added the MPI_COMM_SPLIT_TYPE routine and the communicator split type constant MPI_COMM_TYPE_SHARED.

18. Section 6.6.2 on page 292.
In MPI-2.2, communication involved in an MPI_INTERCOMM_CREATE operation could interfere with point-to-point communication on the parent communicator with the same tag or MPI_ANY_TAG. This interference has been removed in MPI-3.0.

19. Section 6.8 on page 313.
Section 6.8 on page 238. The constant MPI_MAX_OBJECT_NAME also applies for type and window names.

20. Section 7.5.8 on page 346.
MPI_CART_MAP can also be used for a zero-dimensional topologies.

21. Section 7.6 on page 348 and Section 7.7 on page 357.
The following neighborhood collective communication routines were added to support sparse communication on virtual topology grids: MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV, MPI_NEIGHBOR_ALLTOALL, MPI_NEIGHBOR_ALLTOALLV, MPI_NEIGHBOR_ALLTOALLW and the nonblocking variants MPI_INEIGHBOR_ALLGATHER, MPI_INEIGHBOR_ALLGATHERV, MPI_INEIGHBOR_ALLTOALL, MPI_INEIGHBOR_ALLTOALLV, and MPI_INEIGHBOR_ALLTOALLW. The displacement arguments in MPI_NEIGHBOR_ALLTOALLW and MPI_INEIGHBOR_ALLTOALLW were defined as address size integers. In MPI_DIST_GRAPH_NEIGHBORS, an ordering rule was added for communicators created with MPI_DIST_GRAPH_CREATE_ADJACENT.

22. Section 8.7 on page 399 and Section 12.4.3 on page 535.
The use of MPI_INIT, MPI_INIT_THREAD and MPI_FINALIZE was clarified. After MPI is initialized, the application can access information about the execution environment by querying the new predefined info object MPI_INFO_ENV.

23. Section 8.7 on page 399.
Allow calls to MPI_T routines before MPI_INIT and after MPI_FINALIZE.

24. Chapter 11 on page 447.
Substantial revision of the entire One-sided chapter, with new routines for window creation, additional synchronization methods in passive target communication, new one-sided communication routines, a new memory model, and other changes.

25. Section ?? on page ??.
A new MPI Tool Information Interface was added.

The following changes are related to the Fortran language support.

26. Section 2.3 on page 10, and Sections 17.1.1, 17.1.2, 17.1.7 on pages 621, 622, and 637.
The new `mpi_08` Fortran module was introduced.

27. Section 2.5.1 on page 12, and Sections 17.1.2, 17.1.3, 17.1.7 on pages 622, 625, and 637.
Handles to opaque objects were defined as named types within the `mpi_08` Fortran module. The operators `.EQ.`, `.NE.`, `==`, and `/=` were overloaded to allow the comparison of these handles. The handle types and the overloaded operators are also available through the `mpi` Fortran module.

28. Sections 2.5.4, 2.5.5 on pages 15, 16, Sections 17.1.1, 17.1.10, 17.1.11, 17.1.12, 17.1.13 on pages 621, 648, 649, 650, 653, and Sections 17.1.2, 17.1.3, 17.1.7 on pages 622, 625, 637.
Within the `mpi_08` Fortran module, choice buffers were defined as assumed-type and assumed-rank according to Fortran 2008 TS 29113 [41], and the compile-time constant MPI_SUBARRAYS_SUPPORTED was set to `.TRUE.`. With this, Fortran subscript triplets

can be used in nonblocking MPI operations; vector subscripts are not supported in nonblocking operations. If the compiler does not support this Fortran TR 29113 feature, the constant is set to .FALSE..

29. Section 2.6.2 on page 19, Section 17.1.2 on page 622, and Section 17.1.7 on page 637. The ierror dummy arguments are OPTIONAL within the mpi_08 Fortran module.

30. Section 3.2.5 on page 32, Sections 17.1.2, 17.1.3, 17.1.7, on pages 622, 625, 637, and Section 17.2.5 on page 672.
Within the mpi_08 Fortran module, the status was defined as TYPE(MPI_Status). Additionally, within both the mpi and the mpi_f08 modules, the constants MPI_STATUS_SIZE, MPI_SOURCE, MPI_TAG, MPI_ERROR, and TYPE(MPI_Status) are defined. New conversion routines were added: MPI_STATUS_F2F08, MPI_STATUS_F082F, MPI_Status_c2f08, and MPI_Status_f082c, In mpi.h, the new type MPI_F08_status, and the external variables MPI_F08_STATUS_IGNORE and MPI_F08_STATUSES_IGNORE were added.

31. Section 3.6 on page 47.
In Fortran with the mpi module or mpif.h, the type of the buffer_addr argument of MPI_BUFFER_DETACH is incorrectly defined and the argument is therefore unused.

32. Section 4.1 on page 87, Section 4.1.6 on page 110, and Section 17.1.15 on page 653.
The Fortran alignments of basic datatypes within Fortran derived types are implementation dependent; therefore it is recommended to use the BIND(C) attribute for derived types in MPI communication buffers. If an array of structures (in C/C++) or derived types (in Fortran) is to be used in MPI communication buffers, it is recommended that the user creates a portable datatype handle and additionally applies MPI_TYPE_CREATE_RESIZED to this datatype handle.

33. Sections 4.1.10, 5.9.5, 5.9.7, 6.7.4, 6.8, 8.3.1, 8.3.2, 8.3.3, 14.1, 17.1.9 on pages 117, 191, 197, 308, 313, 384, 386, 388, 613, and 639. In some routines, the dummy argument names were changed because they were identical to the Fortran keywords TYPE and FUNCTION. The new dummy argument names must be used because the mpi and mpi_08 modules guarantee keyword-based actual argument lists. The argument name type was changed in MPI_TYPE_DUP, the Fortran USER_FUNCTION of MPI_OP_CREATE, MPI_TYPE_SET_ATTR, MPI_TYPE_GET_ATTR, MPI_TYPE_DELETE_ATTR, MPI_TYPE_SET_NAME, MPI_TYPE_GET_NAME, MPI_TYPE_MATCH_SIZE, the callback prototype definition MPI_Type_delete_attr_function, and the predefined callback function MPI_TYPE_NULL_DELETE_FN; function was changed in MPI_OP_CREATE, MPI_COMM_CREATE_ERRHANDLER, MPI_WIN_CREATE_ERRHANDLER, MPI_FILE_CREATE_ERRHANDLER, and MPI_ERRHANDLER_CREATE. For consistency reasons, INOUBUF was changed to INOUTBUF in MPI_REDUCE_LOCAL, and intracomm to newintracomm in MPI_INTERCOMM_MERGE.

34. Section 6.7.2 on page 299.
It was clarified that in Fortran, the flag values returned by a comm_copy_attr_fn callback, including MPI_COMM_NULL_COPY_FN and MPI_COMM_DUP_FN, are .FALSE. and .TRUE.; see MPI_COMM_CREATE_KEYVAL.

35. Section 8.2 on page 379.
With the mpi and mpi_f08 Fortran modules, MPI_ALLOC_MEM now also supports TYPE(C_PTR) C-pointers instead of only returning an address-sized integer that may be usable together with a non-standard Cray-pointer.

36. Section 17.1.15 on page 653, and Section 17.1.7 on page 637.
Fortran SEQUENCE and BIND(C) derived application types can now be used as buffers in MPI operations.

37. Section 17.1.16 on page 655 to Section 17.1.19 on page 666, Section 17.1.7 on page 637, and Section 17.1.8 on page 638.
The sections about Fortran optimization problems and their solutions were partially rewritten and new methods are added, e.g., the use of the ASYNCHRONOUS attribute. The constant MPI_ASYNC_PROTECTS_NONBLOCKING tells whether the semantics of the ASYNCHRONOUS attribute is extended to protect nonblocking operations. The Fortran routine MPI_F_SYNC_REG is added. MPI-3.0 compliance for an MPI library together with a Fortran compiler is defined in Section 17.1.7.

38. Section 17.1.2 on page 622.
Within the mpi_08 Fortran module, dummy arguments are now declared with INTENT=IN, OUT, or INOUT as defined in the mpi_08 interfaces.

39. Section 17.1.3 on page 625, and Section 17.1.7 on page 637.
The existing mpi Fortran module must implement compile-time argument checking.

40. Section 17.1.4 on page 627.
The use of the mpif.h Fortran include file is now strongly discouraged.

41. Section A.1.1, Table "*Predefined functions*" on page 693, Section A.1.3 on page 700, and Section A.3.4 on page 753.
Within the new mpi_f08 module, all callback prototype definitions are now defined with explicit interfaces PROCEDURE(MPI_...) that have the BIND(C) attribute; user-written callbacks must be modified if the mpi_f08 module is used.

42. Section A.1.3 on page 700.
In some routines, the Fortran callback prototype names were changed from ..._FN to ..._FUNCTION to be consistent with the other language bindings.

## B.4 Changes from Version 2.1 to Version 2.2

1. Section 2.5.4 on page 15.
It is now guaranteed that predefined named constant handles (as other constants) can be used in initialization expressions or assignments, i.e., also before the call to MPI_INIT.

2. Section 2.6 on page 17, and Section 15.2 on page 618.
The C++ language bindings have been deprecated and may be removed in a future version of the MPI specification.

3. Section 3.2.2 on page 27.
MPI_CHAR for printable characters is now defined for C type char (instead of signed

char). This change should not have any impact on applications nor on MPI libraries (except some comment lines), because printable characters could and can be stored in any of the C types char, signed char, and unsigned char, and MPI_CHAR is not allowed for predefined reduction operations.

4. Section 3.2.2 on page 27.
MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are now valid predefined MPI datatypes.

5. Section 3.4 on page 39, Section 3.7.2 on page 51, Section 3.9 on page 77, and Section 5.1 on page 149.
The read access restriction on the send buffer for blocking, non blocking and collective API has been lifted. It is permitted to access for read the send buffer while the operation is in progress.

6. Section 3.7 on page 50.
The Advice to users for IBSEND and IRSEND was slightly changed.

7. Section 3.7.3 on page 55.
The advice to free an active request was removed in the Advice to users for MPI_REQUEST_FREE.

8. Section 3.7.6 on page 67.
MPI_REQUEST_GET_STATUS changed to permit inactive or null requests as input.

9. Section 5.8 on page 176.
"In place" option is added to MPI_ALLTOALL, MPI_ALLTOALLV, and MPI_ALLTOALLW for intracommunicators.

10. Section 5.9.2 on page 184.
Predefined parameterized datatypes (e.g., returned by MPI_TYPE_CREATE_F90_REAL) and optional named predefined datatypes (e.g. MPI_REAL8) have been added to the list of valid datatypes in reduction operations.

11. Section 5.9.2 on page 184.
MPI_(U)INT{8,16,32,64}_T are all considered C integer types for the purposes of the predefined reduction operators. MPI_AINT and MPI_OFFSET are considered Fortran integer types. MPI_C_BOOL is considered a Logical type. MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are considered Complex types.

12. Section 5.9.7 on page 197.
The local routines MPI_REDUCE_LOCAL and MPI_OP_COMMUTATIVE have been added.

13. Section 5.10.1 on page 199.
The collective function MPI_REDUCE_SCATTER_BLOCK is added to the MPI standard.

14. Section 5.11.2 on page 203.
Added in place argument to MPI_EXSCAN.

15. Section 6.4.2 on page 266, and Section 6.6 on page 288.
Implementations that did not implement MPI_COMM_CREATE on intercommunicators will need to add that functionality. As the standard described the behavior of this operation on intercommunicators, it is believed that most implementations already provide this functionality. Note also that the C++ binding for both MPI_COMM_CREATE and MPI_COMM_SPLIT explicitly allow Intercomms.

16. Section 6.4.2 on page 266.
MPI_COMM_CREATE is extended to allow several disjoint subgroups as input if comm is an intracommunicator. If comm is an intercommunicator it was clarified that all processes in the same local group of comm must specify the same value for group.

17. Section 7.5.4 on page 328.
New functions for a scalable distributed graph topology interface has been added. In this section, the functions MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE, the constants MPI_UNWEIGHTED, and the derived C++ class Distgraphcomm were added.

18. Section 7.5.5 on page 335.
For the scalable distributed graph topology interface, the functions MPI_DIST_GRAPH_NEIGHBORS_COUNT and MPI_DIST_GRAPH_NEIGHBORS and the constant MPI_DIST_GRAPH were added.

19. Section 7.5.5 on page 335.
Remove ambiguity regarding duplicated neighbors with MPI_GRAPH_NEIGHBORS and MPI_GRAPH_NEIGHBORS_COUNT.

20. Section 8.1.1 on page 375.
The subversion number changed from 1 to 2.

21. Section 8.3 on page 382, Section 14.2 on page 616, and Annex A.1.3 on page 700.
Changed function pointer typedef names MPI_{Comm,File,Win}_errhandler_fn to MPI_{Comm,File,Win}_errhandler_function. Deprecated old "_fn" names.

22. Section 8.7.1 on page 405.
Attribute deletion callbacks on MPI_COMM_SELF are now called in LIFO order. Implementors must now also register all implementation-internal attribute deletion callbacks on MPI_COMM_SELF before returning from MPI_INIT/MPI_INIT_THREAD.

23. Section 11.3.4 on page 470.
The restriction added in MPI 2.1 that the operation MPI_REPLACE in MPI_ACCUMULATE can be used only with predefined datatypes has been removed. MPI_REPLACE can now be used even with derived datatypes, as it was in MPI 2.0. Also, a clarification has been made that MPI_REPLACE can be used only in MPI_ACCUMULATE, not in collective operations that do reductions, such as MPI_REDUCE and others.

24. Section 12.2 on page 523.
Add "*" to the query_fn, free_fn, and cancel_fn arguments to the C++ binding for MPI::Grequest::Start() for consistency with the rest of MPI functions that take function pointer arguments.

25. Section 13.5.2 on page 591, and Table 13.2 on page 592.
MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_COMPLEX, MPI_C_FLOAT_COMPLEX, MPI_C_DOUBLE_COMPLEX, MPI_C_LONG_DOUBLE_COMPLEX, and MPI_C_BOOL are added as predefined datatypes in the external32 representation.

26. Section 17.2.7 on page 677.
The description was modified that it only describes how an MPI implementation behaves, but not how MPI stores attributes internally. The erroneous MPI-2.1 Example 16.17 was replaced with three new examples 17.13, 17.14, and 17.15 on pages 678-679 explicitly detailing cross-language attribute behavior. Implementations that matched the behavior of the old example will need to be updated.

27. Annex A.1.1 on page 685.
Removed type MPI::Fint (compare MPI_Fint in Section A.1.2 on page 698).

28. Annex A.1.1 on page 685. Table *Named Predefined Datatypes*.
Added MPI_(U)INT{8,16,32,64}_T, MPI_AINT, MPI_OFFSET, MPI_C_BOOL, MPI_C_FLOAT_COMPLEX, MPI_C_COMPLEX, MPI_C_DOUBLE_COMPLEX, and MPI_C_LONG_DOUBLE_COMPLEX are added as predefined datatypes.

## B.5 Changes from Version 2.0 to Version 2.1

1. Section 3.2.2 on page 27, and Annex A.1 on page 685.
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, and Annex A.1 on page 685.
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 32.
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 87.
General rule about derived datatypes: Most datatype constructors have replication count or block length arguments. Allowed values are non-negative integers. If the value is zero, no elements are generated in the type map and there is no effect on datatype bounds or extent.

5. Section 4.3 on page 144.
MPI_BYTE should be used to send and receive data that is packed using MPI_PACK_EXTERNAL.

6. Section 5.9.6 on page 195.
If comm is an intercommunicator in MPI_ALLREDUCE, then both groups should pro-

vide 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 256.
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 297.
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_YYY_GET_ATTR, MPI_YYY_SET_ATTR, MPI_YYY_DELETE_ATTR, or MPI_YYY_FREE_KEYVAL. To do so, it is necessary to maintain, with each keyval, information on the type of the associated user function. (*End of advice to implementors.*)

9. Section 6.8 on page 313.
In MPI_COMM_GET_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_OBJECT_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_OBJECT_NAME.

10. Section 7.4 on page 322.
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 324.
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 326.
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 326.
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 7.5.5 on page 335.
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 7.5.5 on page 335.
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 7.5.5 on page 335.
In MPI_CART_COORDS: If comm is associated with a zero-dimensional Cartesian topology, coords will be unchanged.

17. Section 7.5.6 on page 343.
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 7.5.7 on page 345.
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.

18.1. Section 8.1.1 on page 375.
The subversion number changed from 0 to 1.

19. Section 8.1.2 on page 377.
In MPI_GET_PROCESSOR_NAME: In C, a null character is additionally stored at name[resultlen]. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME-1. In Fortran, name is padded on the right with blank characters. resultlen cannot be larger then MPI_MAX_PROCESSOR_NAME.

20. Section 8.3 on page 382.
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 399, 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 441.

22. Section 8.7 on page 399.
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 409.
An implementation must support info objects as caches for arbitrary (key, value) pairs, regardless of whether it recognizes the key. Each function that takes hints in the form of an MPI_Info must be prepared to ignore any key it does not recognize. This description of info objects does not attempt to define how a particular function should react if it recognizes a key but not the associated value. MPI_INFO_GET_NKEYS, MPI_INFO_GET_NTHKEY, MPI_INFO_GET_VALUELEN, and MPI_INFO_GET must retain all (key,value) pairs so that layered functionality can also use the Info object.

24. Section 11.3 on page 463.
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 463.
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 470.
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 551.
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 551.
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 554.
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 591.
The bias of 16 byte doubles was defined with 10383. The correct value is 16383.

31. MPI-2.2, Section 16.1.4 (Section was removed in MPI-3.0).
In the example in this section, the buffer should be declared as `const void* buf`.

32. Section 17.1.9 on page 639.
About MPI_TYPE_CREATE_F90_XXX:

*Advice to implementors.* An application may often repeat a call to MPI_TYPE_CREATE_F90_XXX with the same combination of (XXX,p,r). The application is not allowed to free the returned predefined, unnamed datatype handles. To prevent the creation of a potentially huge amount of handles, the MPI implementation should return the same datatype handle for the same (

REAL/COMPLEX/INTEGER,p,r) combination. Checking for the combination ( p,r) in the preceding call to MPI_TYPE_CREATE_F90_XXX and using a hash-table to find formerly generated handles should limit the overhead of finding a previously generated datatype with same combination of (XXX,p,r). (*End of advice to implementors.*)

33. Section A.1.1 on page 685.
MPI_BOTTOM is defined as `void * const MPI::BOTTOM`.

# Bibliography


[1] V. Bala and S. Kipnis. Process groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2

[2] V. Bala, S. Kipnis, L. Rudolph, and Marc Snir. Designing efficient, scalable, and portable collective communication libraries. Technical report, IBM T. J. Watson Research Center, October 1992. Preprint. 1.2

[3] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++: Issues and Features. In *OON-SKI '94*, page in press, 1994. 6.1

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6):88–95, June 1993. 1.2

[5] Luc Bomans and Rolf Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990. 1.2

[6] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *IJHPCN*, 1(1/2/3):91–99, 2004. 11.7

[7] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993. 13.1

[8] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report TM-ANL–92/17, Argonne National Laboratory, 1992. 1.2

[9] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994. Also Argonne National Laboratory Mathematics and Computer Science Division preprint P362-0493. 1.2

[10] Robin Calkin, Rolf Hempel, Hans-Christian Hoppe, and Peter Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, April 1994. 1.2

[11] S. Chittor and R. J. Enbody. Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers. In *Proceedings of the 1990 Supercomputing Conference*, pages 647–656, 1990. 7.1

[12] S. Chittor and R. J. Enbody. Predicting the effect of mapping on the communication performance of large multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II (Software)*, pages II–1 – II–4, 1991. 7.1

[13] Parasoft Corporation. Express version 1.0: A communication environment for parallel computers, 1988. 1.2

[14] Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*. Springer, 2011. 16, 36

[15] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38. 13.1

[16] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective communicator creation in MPI. In Cotronis et al. [14], pages 282–291. 6.4.2

[17] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–75, April 1993. 1.2

[18] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993. 1.2

[19] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Concepts*, June 1991. 1.2

[20] Edinburgh Parallel Computing Centre, University of Edinburgh. *CHIMP Version 1.0 Interface*, May 1992. 1.2

[21] D. Feitelson. Communicators: Object-based multiparty interactions for parallel programming. Technical Report 91-12, Dept. Computer Science, The Hebrew University of Jerusalem, November 1991. 6.1.2

[22] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994. 1.3

[23] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report, 1995. `http://www.mpi-forum.org`. 1.3

[24] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994. 10.1

[25] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. PICL: A portable instrumented communications library, C reference manual. Technical Report TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990. 1.2

[26] D. Gregor, T. Hoefler, B. Barrett, and A. Lumsdaine. Fixing probe for multi-threaded MPI applications. Technical Report 674, Indiana University, Jan. 2009. 3.8.2

[27] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993. 1.2

[28] Michael Hennecke. A Fortran 90 interface to MPI version 1.1. Technical Report Internal Report 63/96, Rechenzentrum, Universität Karlsruhe, D-76128 Karlsruhe, Germany, June 1996. Available via world wide web from `http://www.uni-karlsruhe.de/~Michael.Hennecke/Publications/#MPI_F90`. 17.1.3

[29] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient MPI support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010. 3.8.1, 3.8.2

[30] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624–633, Sep. 2007. 5.12

[31] T. Hoefler, F. Lorenzen, and A. Lumsdaine. Sparse non-blocking collectives in quantum mechanical calculations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 55–63. Springer, Sep. 2008. 7.6

[32] T. Hoefler and A. Lumsdaine. Message progression in parallel computing — to thread or not to thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008. 5.12

[33] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007. 5.12

[34] T. Hoefler, M. Schellmann, S. Gorlatch, and A. Lumsdaine. Communication optimization for medical image reconstruction algorithms. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume LNCS 5205, pages 75–83. Springer, Sep. 2008. 5.12

[35] T. Hoefler and J. L. Traeff. Sparse collective operations for MPI. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 Workshop*, May 2009. 7.6

[36] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI — common practice, issues, and extensions. In Cotronis et al. [14], pages 345–355. 6.4.2

[37] Institute of Electrical and Electronics Engineers, New York. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985. 13.5.2

[38] International Organization for Standardization, Geneva, ISO 8859-1:1987. *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, 1987. 13.5.2

[39] International Organization for Standardization, Geneva, ISO/IEC 9945-1:1996(E). *Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language]*, December 1996. 12.4, 13.2.1

[40] International Organization for Standardization, Geneva, ISO/IEC 1539-1:2010. *Information technology – Programming languages – Fortran – Part 1: Base language*, November 2010. 17.1.1, 17.1.2

[41] International Organization for Standardization, ISO/IEC/SC22/WG5 (Fortran), Geneva, TS 29113. *TS on further interoperability with C*, 2012. http://www.nag.co.uk/sc22wg5/, successfully balloted DTS at ftp://ftp.nag.co.uk/sc22wg5/N1901-N1950/N1917.pdf. 17.1.1, 17.1.1, 17.1.2, 17.1.7, 28

[42] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1993. 4.1.4

[43] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994. 13.1

[44] O. Krämer and H. Mühlenbein. Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225, 1989. 7.1

[45] S. J. Lefflet, R. S. Fabry, W. N. Joy, P. Lapsley, S. Miller, and C. Torek. An advanced 4.4BSD interprocess communication tutorial, Unix programmer's supplementary documents (PSD) 21. Technical report, Computer Systems Research Group, Depertment of Electrical Engineering and Computer Science, University of California, Berkeley, 1993. Also available at `http://www.netbsd.org/Documentation/lite2/psd/`. 10.5.5

[46] nCUBE Corporation. *nCUBE 2 Programmers Guide, r2.0*, December 1990. 1.2

[47] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992. 13.1

[48] William J. Nitzberg. *Collective Parallel I/O*. PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995. 13.1

[49] *4.4BSD Programmer's Supplementary Documents (PSD)*. O'Reilly and Associates, 1994. 10.5.5

[50] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390. ACM Press, 1988. 1.2

[51] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995. 13.1

[52] A. Skjellum and A. Leung. Zipcode: a portable multicomputer communication library atop the reactive kernel. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 767–776. IEEE Press, 1990. 1.2, 6.1.2

[53] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. The Zipcode message passing system. Technical report, Lawrence Livermore National Laboratory, September 1992. 1.2

[54] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, *Proceedings of the Scalable Parallel Libraries Conference*, pages 166–173. IEEE Computer Society Press, October 1993. 6.1

[55] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) Library. Technical Report MSU-940722, Mississippi State University — Dept. of Computer Science, April 1994. `http://www.erc.msstate.edu/mpi/mpix.html`. 5.2.2

[56] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. *Parallel Computing*, 20(4):565–596, April 1994. 6.1.2, 6.5.6

[57] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996. 13.1

[58] *The Unicode Standard, Version 2.0*. Addison-Wesley, 1996. ISBN 0-201-48345-9. 13.5.2

[59] D. Walker. Standards for message passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 1.2

# General Index

This index lists mainly terms of the MPI specification. The underlined page numbers refer to the definitions or parts of the definition of the terms. Bold face numbers mark section titles.

# Examples Index

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI function that they are demonstrating. MPI functions listed in all capital letter are Fortran examples; MPI functions listed in mixed case are C examples.

# MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

# MPI Declarations Index

This index refers to declarations needed in 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 Callback Function Prototype Index

This index lists the C typedef names for callback routines, such as those used with attribute caching or user-defined reduction operations. Fortran example prototypes are given near the text of the C name.

# MPI Function Index

The underlined page numbers refer to the function definitions.